EXERCICES ET PROBLÈMES D'ALGORITHMIQUE

- **Rappels de cours**
- **Exercices et problèmes avec corrigés détaillés**
- **Solutions en pseudo code et en langage C**

Nicolas Flasque
Enseignant mathématiques et informatique, EFREI

Helen Kassel
Enseignant mathématiques et informatique, EFREI

Franck Lepoivre
Enseignant-chercheur

Boris Velikson
Enseignant mathématiques et informatique, EFREI



Illustration de couverture : *digitalvision*®

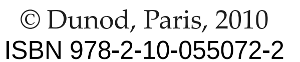

ISBN 978-2-10-055072-2

Table des matières

AVANT-PROPOS

Cet ouvrage s'adresse aux élèves des écoles d'ingénieurs, aux élèves d'IUT, de DUT, de BTS, aux auditeurs des organismes de formation continue et aux autodidactes qui souhaitent se doter de bases pratiques et théoriques en algorithmique. Le niveau de maîtrise attendu correspond à la seconde année de licence.

MODE D'EMPLOI

Un contenu construit pour aller directement à l'essentiel

Cet ouvrage de travaux dirigés d'algorithmique est construit pour aller directement à l'essentiel sans faire d'impasse sur ce qui est important, ni se disperser dans ce qui viendra à point nommé dans les étapes de votre apprentissage.

Simple d'accès, il contient les chapitres classiques d'une introduction à l'algorithmique, avec notamment les structures séquentielles, arborescentes, et les automates.

Chaque chapitre débute avec un rappel de cours d'une vingtaine de pages suivi des énoncés et corrigés des exercices et problèmes.

Pour compléter cette structure classique, un chapitre introductif résume les bases minimales de la programmation informatique.

Les corrigés sont donnés sous la forme suivante :

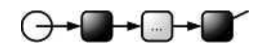
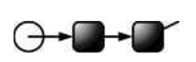
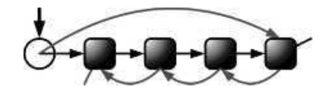
- une éventuelle étude des stratégies de résolution du problème posé (si celui-ci est complexe), accompagnée de schémas descriptifs de principe ;
- une spécification en langage algorithmique (pseudo code) de la ou des solutions envisagées ;
- une éventuelle proposition de réalisation en C99 des solutions proposées.

Des schémas intuitifs

Les schémas descriptifs de principe facilitent la compréhension des principes de fonctionnement des algorithmes proposés.

La liste suivante vous sera utile notamment pour interpréter les schémas du second chapitre.

			
Une place quelconque	Un pointeur sur une place non vide (et donc le début d'une liste de places)	Une place pointant sur la suivante (place intermédiaire)	Une place intermédiaire contenant l'élément 6

 La liste vide (≡ un pointeur ne pointant sur rien)	 Une place terminale (par composition)	 Un singleton (liste à un seul élément)	 Une liste à éléments multiples
 Le cas particulier du couple (liste à deux éléments)	 Représentation des modifications effectuées (pointillés (après) vs. traits pleins (avant))		

Un plan de travail qui peut être adapté

Si vous débutez et n'avez jamais écrit le moindre programme informatique de votre vie, la lecture du premier chapitre vous sera nécessaire. Sinon, elle n'est pas indispensable, sauf éventuellement comme référence pour le langage algorithmique utilisé dans les corrigés.

Si vous démarrez avec quelques notions de programmation, les deux chapitres sur les structures séquentielles et arborescentes vous donneront les bases nécessaires pour raisonner en termes algorithmiques et aborder par la suite des structures et algorithmes plus complexes, bâtis sur ces éléments de bases.

Enfin, quel que soit votre niveau, le dernier chapitre sur les automates vous sensibilisera sur les fondements mathématiques de l'algorithmique, notamment des logiques d'exécution.

Avec les structures séquentielles et les approches itératives, les structures arborescentes et les approches récursives, et enfin, avec les automates et les logiques générales d'exécution, vous munirez votre arc de trois cordes essentielles pour aborder la suite de votre apprentissage.

À PROPOS DES AUTEURS

- **Nicolas Flasque**
  Ingénieur IIE depuis 1992 et docteur en informatique depuis 2001. Après avoir travaillé une année en tant que responsable logiciel sur les systèmes embarqués automobiles, il reprend ses études et obtient un doctorat de l'université de Caen sur la reconnaissance de vaisseaux sanguins pour l'imagerie médicale. En poste à l'EFREI depuis septembre 2001, il enseigne l'algorithmique ainsi que la programmation dans des langages nécessitant des approches différentes (C, C++, C#, Java).

- **Helen Kassel**
  De double formation en mathématiques (DEA obtenu en Russie) et en informatique (DEA obtenu en France), elle a enseigné l'informatique en Russie, aux États-Unis et en France. Elle possède également une expérience du travail en entreprise en tant qu'ingénieur en informatique. Enseignant en informatique et en mathématiques à l'EFREI depuis plus de dix ans, elle est actuellement le chef du département mathématiques/informatique.
- **Franck Lepoivre**
  Diplômé ingénieur de l'ISEP en 1995, il évolue dans les entreprises de nouvelles technologies en tant que consultant IT (coauteur de XML & Java, Eyrolles 2000) puis directeur marketing produit (prix technologia ANVAR et 01 Informatique pour Kelua Kawana en 2002). En 2004, il lance *reciproCity* pour porter l'analyse sociologique dans le domaine de l'intelligence économique. En 2007, il lance *Pepper Labs* pour porter les mathématiques appliquées et algorithmique vers les entreprises et leur problématiques métier (modélisation et prototypage d'outils d'analyse complexe, notamment dans les domaines du marketing et des neurosciences appliquées). Il intervient à l'EFREI en algorithmique et structures de données, théorie des langages et techniques de compilation, théorie des graphes, aide à la décision et algorithmique numérique.
- **Boris Velikson**
  Diplômé de Ph.D. en Physique théorique aux États-Unis après un Bac+5 en Russie, il a travaillé comme chercheur en théorie des champs quantiques et puis en biophysique, dans le domaine de modélisation de grosses molécules biologiques sur ordinateur. Depuis plusieurs années, il travaille comme enseignant en mathématiques, en statistique et en informatique, dans quelques établissements de la région parisienne, à des niveaux très différents, en français et en anglais.

REMERCIEMENTS

Nous remercions nos étudiants de l'EFREI, sans qui l'élaboration de ce contenu n'aurait pu trouver le juste diapason pédagogique. C'est par la somme de nos interactions qu'émergent et s'améliorent nos contenus d'apprentissage par la pratique.

Nous remercions notre éditeur, Jean-Luc Blanc, qui nous a donné la chance de produire ce cahier sur la base de nos existants pédagogiques dans le cadre de la collection *Exercices & Problèmes* où il trouve une place cohérente par rapport à d'autres ouvrages de mathématiques appliquées.

Nous remercions nos familles et nos amis, pour avoir toléré ce temps supplémentaire que nous leur avons soustrait, et pour leur soutien pourtant indéfectible.

INTRODUCTION

QU'EST-CE QUE L'ALGORITHMIQUE ?

Un **problème** est un questionnement qui appelle une **solution**. Mais existe-t-il seulement une solution ?

Tout problème en induit deux autres, deux questions préalables à toute tentative de résolution, et dont les réponses ne vont pas toujours d'elles-mêmes, et ne sont pas nécessairement affirmatives. Ce sont les deux questions de **décidabilité** :

- La première est celle de la **décidabilité logique ou théorique** : ce problème, est-il soluble ? Construire la réponse relève des mathématiques pures et non pas de l'art algorithmique à proprement parler. Répondre à cette question par la négative peut éviter la vaine recherche d'une réponse à la seconde.
- La certitude d'une possibilité de résolution acquise, se pose la seconde question de la **décidabilité algorithmique ou pratique** : comment trouver la solution ?

Résoudre en pratique un problème théoriquement soluble, c'est concevoir et opérer une méthode de **raisonnement** qui, partant d'un énoncé qualitatif et quantitatif, permet de construire en un nombre fini d'étapes, l'énoncé de sa solution.

Un **algorithme** est la description d'une telle méthode de raisonnement comme succession d'étapes élémentaires et intermédiaires de résolution, ce qu'on appelle communément un **calcul**. Ainsi un algorithme se conçoit-il naturellement comme une décomposition d'un problème en sous-problèmes plus simples, individuellement « faciles » à résoudre et dont la composition donne la solution, plus complexe, du problème principal.

Mais est-ce la meilleure façon de procéder ?

Si décrire un algorithme, signifie décrire une méthode de raisonnement (un programme) qui détermine la solution d'un problème en un nombre fini d'étapes de calcul, il se peut que le temps nécessaire à ce calcul place le résultat final hors de portée.

C'est ici qu'interviennent les notions d'**équifinalité**[1], notion prélevée sur le vocabulaire stratégique, et de **complexité algorithmique**.

Une méthode de résolution n'est jamais unique, et les stratégies alternatives, c'est-à-dire les différentes façons d'aboutir au même résultat ne sont pas tactiquement égales. Certaines sont plus

1. Notion prélevée sur le vocabulaire cybernétique et stratégique, l'*équifinalité* traduit la possibilité pour un système d'atteindre un même but par différents chemins, i.e. une seule stratégie (le but), mais plusieurs tactiques pour réaliser la stratégie.

coûteuses que d'autres, en termes de ressources temps, en termes de ressources mnémoniques mobilisées.

Savoir évaluer, avant de l'exécuter, l'efficacité d'un algorithme, chercher à systématiquement minimiser ce coût au moment de le concevoir, c'est assurément ce qui pose l'**algorithmique** comme un art.

COMMENT DEVENIR ALGORITHMICIEN ?

L'apprentissage traditionnel de l'algorithmique élude les aspects les plus formels et sophistiqués de la décidabilité, de la calculabilité et de la complexité, qui s'ils sont fondamentaux, ne sont pas nécessairement faciles d'accès.

On commence généralement l'apprentissage par la pratique de la programmation à l'aide d'un langage simple, puis dans un second temps, on prend du recul par rapport à cette première approche, pour découvrir les aspects les plus généraux des structures de données et des algorithmes standards. Enfin, on aborde les éléments plus « mathématiques » de la complexité après en avoir « ressenti » la réalité par l'expérience programmatique.

Une étape majeure, qui fera la différence entre programmeur et algorithmicien, consistera à prendre de la distance avec la programmation, et à se représenter dans toute leur généralité, les schémas algorithmiques, indépendamment de tout langage d'implantation. L'influence du paradigme de programmation spécifique du premier langage appris est souvent le frein qui empêche d'aborder l'algorithmique selon la bonne perspective.

À l'autre extrémité du spectre de progression, destiné à l'ingénieur en informatique accompli, un ouvrage tel que le *TAOCP*[1] de Donald E. Knuth qui représente la « quintessence » de l'art algorithmique, est un ouvrage radicalement indigeste pour qui fait ses premiers pas en informatique.

QU'EST-CE QU'UN ALGORITHME ?

Selon l'*Encyclopedia Universalis* un algorithme est la « *spécification d'un schéma de calcul, sous forme d'une suite [finie] d'opérations élémentaires obéissant à un enchaînement déterminé* ».

On connaît depuis l'antiquité des algorithmes sur les nombres, comme par exemple l'**algorithme d'Euclide** qui permet de calculer le p.g.c.d. de deux nombres entiers.

Pour le traitement de l'information, on a développé des algorithmes opérant sur des données non numériques : les **algorithmes de tri**, qui permettent par exemple de ranger par ordre alphabétique une suite de noms, les algorithmes de recherche d'une chaîne de caractères dans un texte, ou les algorithmes d'ordonnancement, qui permettent de décrire la coordination entre différentes tâches, nécessaire pour mener à bien un projet.

1. *TAOCP*, *The Art of Computer Programming*, la « Bible » de l'algorithmique en quatre volumes par Donald E. Knuth, professeur émérite de Stanford et inventeur de TEX. TAOCP est une encyclopédie algorithmique plus proche des mathématiques pures que de la programmation informatique.

Un programme destiné à être exécuté par un ordinateur, est la plupart du temps, la description d'un algorithme dans un langage accepté par cette machine.

Définissons plus formellement le concept :

- Un algorithme décrit un traitement sur un certain nombre, fini, de données.
- Un algorithme est la composition d'un ensemble fini d'étapes, chaque étape étant formée d'un nombre fini d'opérations dont chacune est :
  - définie de façon rigoureuse et non ambiguë ;
  - effective, c'est-à-dire pouvant être effectivement réalisée par une machine : cela correspond à une action qui peut être réalisée avec un papier et un crayon en un temps fini ; par exemple la division entière est une opération effective, mais pas la division avec un nombre infini de décimales.

Quelle que soit la donnée sur laquelle on travaille, un algorithme doit toujours se terminer après un nombre fini d'opérations, et fournir un résultat.

Conception d'un algorithme

La conception d'un algorithme un peu compliqué se fait toujours en plusieurs étapes qui correspondent à des raffinements successifs. La première version de l'algorithme est autant que possible indépendante d'une implémentation particulière.

En particulier, la représentation des données n'est pas fixée.

À ce premier niveau, les données sont considérées de manière abstraite : on se donne une notation pour les décrire ainsi que l'ensemble des opérations qu'on peut leur appliquer et les propriétés de ces opérations. On parle alors de **type abstrait de données**. La conception de l'algorithme se fait en utilisant les opérations du type abstrait.

Pour résoudre des problèmes nous allons appliquer une **démarche descendante** : on se donne la définition des types de données (on dit encore leur spécification), et on conçoit l'algorithme à ce niveau.

On donne ensuite une représentation concrète des types et des opérations, qui peut être encore un type abstrait, et ceci jusqu'à obtenir un programme exécutable.

Notion de structure de données

Une structure de données est un ensemble organisé d'informations reliées logiquement, ces informations pouvant être traitées collectivement ou individuellement.

L'exemple le plus simple : le tableau monodimensionnel (un vecteur) est constitué d'un certain nombre de composantes de même type.

On peut effectuer des opérations sur chaque composante prise individuellement mais on dispose aussi d'opérations globales portant sur le vecteur considéré comme un seul objet.

Une structure de données est caractérisée par ses composantes et leur arrangement mais surtout par son mode de traitement.

Ainsi deux structures ayant les mêmes composantes, les mêmes arrangements comme les PILES et FILES d'ATTENTE sont considérées comme différentes car leurs modes d'exploitation sont fondamentalement différents.

1 Les bases de la programmation

1.1 Les types de données

Un **type** en algorithmique est une information permettant de traduire les valeurs depuis une représentation binaire (celle de l'ordinateur) vers une autre représentation plus adaptée à leur programmation dans un langage évolué. Cette notion est tellement importante que toute valeur a forcément un type. Le rôle du type est d'assurer cette traduction en indiquant quelle place en mémoire occupe la valeur et quelle est la technique de codage utilisée.

Nous distinguons quatre types élémentaires en algorithmique :

- Le type `entier` sera utilisé pour stocker des valeurs entières, positives ou négatives. Un entier occupe quatre octets (32 bits) en mémoire.
- Le type `réel` sera utilisé pour stocker les nombres à virgule. Un réel occupe huit octets (64 bits) en mémoire.
- Le type `caractère` sera utilisé pour stocker les caractères. Un caractère occupe un octet (8 bits) en mémoire.
- Le type `booléen` sera utilisé pour stocker les valeurs de type vrai/faux. Un booléen occupe un octet (8 bits) en mémoire.



Attention au type dit « réel ». En effet, un ordinateur ne stocke ses valeurs que sur une place limitée, il ne stocke donc qu'un nombre limité de décimales après la virgule. Les valeurs de type réel en algorithmique ne sont donc que des valeurs approchées de leur version mathématique !

Remarque

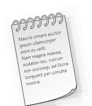
Le type utilisé pour stocker des caractères est un peu particulier, car un caractère est en fait un nombre entier ! L'ordinateur utilise une table de correspondance qui associe une valeur entière (un code) à un caractère qu'il s'agit de manipuler, c'est-à-dire, la plupart du temps, pour l'afficher à l'écran. Cette table de correspondance se nomme la table de symboles (table ASCII, Unicode).

Pour la table ASCII, un caractère est stocké dans un octet (un groupement de 8 bits), la valeur entière peut donc aller de 0 à 255. Dans le tableau ne sont présentes que les valeurs de 32 à 127 : en deçà de 32, il s'agit de caractères non imprimables, au-delà de 127, ce sont des caractères optionnels, qui sont adaptés à un type de clavier ou de langue particulier, notamment les caractères accentués (é, à, è, ô, â, etc.).

1.2 LES VARIABLES

Une **variable** est une donnée qu'un programme peut manipuler. Tout variable possède :

- Un **type** (entier, réel, caractère ou booléen).
- Un **nom** ou identificateur que l'utilisateur choisit ; il permet au programme de reconnaître quelle donnée il doit manipuler.
- Une **valeur** qui peut évoluer au cours du programme, mais qui doit respecter le type.



Une variable dont le type est entier ne pourra donc jamais contenir de valeur à virgule.

L'identificateur ou nom de la variable peut être quelconque, mais doit respecter les critères suivants :

- un identificateur commence toujours par une lettre minuscule ;
- à l'exception du premier caractère, il peut contenir : des lettres, des chiffres, et le symbole '_' (souligné ou *underscore*) ;
- les majuscules et les minuscules sont des lettres différentes : les identificateurs `toto` et `Toto` sont différents ;
- le nom de variable doit avoir une relation avec le rôle de cette variable et être compréhensible.

1.3 QUELQUES ÉLÉMENTS DE SYNTAXE POUR LE LANGAGE ALGORITHMIQUE

Pour écrire correctement un programme en langage algorithmique, il faut fournir certaines informations à l'ordinateur : le mot *programme* suivi du *nom du programme*, indique le nom du programme ainsi que son point de départ.

Avant d'utiliser une variable dans un programme, il faut la **définir**, c'est-à-dire indiquer le mot `VAR`, puis le nom de la variable et enfin son type précédé de ':'.

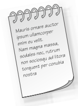
Une variable s'appelant `taux`, et dont le type est `réel`, doit être définie de la manière suivante :

```
VAR taux : réel
```

Cette définition crée une variable nommée taux dans laquelle peuvent être stockés des nombres à virgule.

Quelques exemples de définition de variables

`VAR solution_equation : réel` définit une variable nommée `solution_equation` dont le type est réel ;
`VAR val_1 : entier` définit une variable nommée `val_1` dont le type est entier ;
`VAR lettre : caractère` définit une variable nommée lettre dont le type est caractère.



Il est possible de définir plusieurs variables d'un même type en spécifiant le nom du type, puis la liste des noms de variables séparés par des virgules ','. Ainsi, les définitions suivantes sont strictement équivalentes : `VAR val_a : entier, VAR val_b : entier, VAR val_c : entier ⇔ VAR val_a, val_b, val_c : entier`

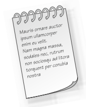
1.4 Opérations et opérateurs de base

1.4.1 Affectation

L'opération d'**affectation** permet de donner (ou d'affecter, d'où son nom) une valeur à une variable. Sa syntaxe est la suivante :

```
nom_de_variable ← valeur_à_affecter
```

Le symbole ← indiquant le sens de l'affectation.

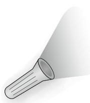


La valeur d'une variable qui n'a pas subi d'affectation est aléatoire. Elle est représentée par un point d'interrogation.

1.4.2 Constantes

Il est possible d'affecter des valeurs numériques, appelées **constantes**, dans une variable. Comme toute valeur, une constante est typée, et ce type a une influence sur la syntaxe :

- **Constantes de type entier** : il suffit juste d'écrire la valeur en base dix, cette valeur peut être positive ou négative. La variable recevra alors la valeur choisie.
- **Constantes de type réel** : elles sont écrites sous la forme mantisse – exposant, c'est-à-dire la notation scientifique avec les puissances de dix, utilisée par les calculatrices. La virgule est représentée par un point (notation anglo-saxonne), et l'exposant, qui est un nombre positif ou négatif, est précédé du symbole E. Il est possible de ne pas indiquer :
  - l'exposant lorsque celui-ci est nul ;
  - le signe + devant l'exposant si celui-ci est positif ;
  - la partie décimale d'un nombre si celle-ci est nulle, par contre on fera toujours figurer le point décimal.



Constantes de type caractère – Il est possible d'affecter un entier à un caractère en utilisant une constante entière ou en indiquant le caractère souhaité entouré de simples guillemets. Ces simples guillemets se lisent alors : "code ASCII de".

Exemple de programme d'affectation

```
programme affectations
VAR a : entier
VAR x : réel
VAR ma_var : caractère
a ← -6
```

```
x ← 6.022E+23
ma_var ← 'Y'   // équivaut à ma_var ← 101
               // car le code ASCII de Y vaut 101
```

1.4.3 Opérateurs arithmétiques et expressions

Il est également intéressant de pouvoir effectuer des calculs et d'en affecter le résultat à une variable. Nous retrouvons sans surprise les opérateurs arithmétiques les plus classiques :

- Addition : +
- Soustraction : -
- Multiplication : *
- Division : /
- Modulo : %

Ces opérateurs agissent avec des constantes et/ou des variables dont la valeur est utilisée pour le calcul à effectuer.

Avec ces opérateurs, les variables et les constantes, il est possible d'écrire ce que l'on appelle des **expressions** : une expression est une suite d'opérateurs et de termes qui est compréhensible et que l'on peut calculer.

(x+3)/2–(4–x)*7 est une expression, car on peut appliquer les opérations, mais 2)+)*5 × 8/(–9 n'est pas une expression, bien que tout ce qui la compose soit des opérations, des termes, et des parenthèses !

Lors du traitement de l'affectation générique : variable ← expression, l'ordinateur calcule dans un premier temps la valeur numérique de l'expression fournie à droite de l'opérateur d'affectation puis range dans un second temps cette valeur calculée dans la variable qui se trouve à gauche de l'opérateur d'affectation.

1.4.4 Opérateurs d'entrée/sortie

Les opérations que nous venons d'aborder permettent juste de faire des calculs, mais ne permettent pas encore de visualiser les résultats ou d'afficher du texte, ou encore de faire des saisies au clavier. Pour cela, nous utiliserons les commandes `AFFICHER` et `SAISIR` :

- `AFFICHER` sert, comme son nom l'indique, à afficher du texte ou les valeurs des variables. On utilise afficher, suivi entre parenthèses des différents éléments à faire apparaître à l'écran. Ces éléments sont soit du texte brut écrit entre doubles guillemets, soit une expression. Dans le cas de texte brut, ce dernier apparaît tel quel à l'écran. Dans le cas d'une expression, c'est la valeur numérique du calcul de cette expression qui est affichée. Les éléments successifs à afficher sont séparés par une virgule.

Exemple

```
VAR t : réel    // définition de la variable entière t
t ← 2.421
AFFICHER  ("t vaut : ", t, " !")   // cette instruction fera apparaître
// à l'écran le message suivant : t vaut 2.421 !
```

- `SAISIR` permet d'initialiser une variable à partir d'une saisie faite au clavier. On utilise `saisir`, suivi entre parenthèses du nom de la variable que l'on veut saisir. L'instruction `saisir` a pour seul effet d'attendre que l'utilisateur entre une valeur au clavier et la valide en appuyant sur la touche « entrée » ou « enter » ; aucun message ne s'affiche pour indiquer à l'utilisateur ce qu'il doit faire ; c'est donc au programmeur de penser à afficher un message pour indiquer qu'une saisie doit être faite !

Exemple

```
VAR a : entier    // définition de la variable entière a
SAISIR(a)    // saisie de la variable
             // l'utilisateur doit entrer une valeur au clavier
             // et la valider par la touche 'entrée'
```



Contrairement à l'opérateur `AFFICHER`, on ne peut saisir que des variables avec `SAISIR`.

1.5 Structure de contrôle

Les **structures de contrôle** (branchements conditionnels et boucles) permettent à un programme de ne pas être purement séquentiel (chaîne linéaire d'instructions).

L'exemple le plus simple à traiter est celui de la résolution d'une équation du second degré dans $\mathbb{R}$ (qui a deux, une ou aucune solution) en fonction de la valeur des coefficients. Le programme qui doit résoudre ce problème devra donc adapter son comportement en fonction des valeurs prises par certaines variables (notamment le discriminant de l'équation).

1.5.1 Conditions et tests

Une **condition** est une expression dont le résultat n'est pas une valeur numérique, mais VRAI ou FAUX, qui sont les deux éléments de l'algèbre dite booléenne ou encore logique. Le calcul booléen respecte un certain nombre de règles, qui sont très simples : cela revient à résoudre des petits problèmes de logiques. Les opérateurs booléens sont également très simples à comprendre et à manipuler.

Les opérateurs booléens de comparaison sont présentés dans le tableau 1.1 (page suivante).

Grâce à ces opérateurs, il est possible d'écrire des conditions élémentaires pour réaliser des tests simples.

1.5.2 Exécution conditionnelle d'instructions

Structure SI...ALORS

La structure de contrôle `SI...ALORS` permet d'exécuter des instructions en fonction de la valeur d'une condition (qui n'est autre que le résultat d'un test).

Tableau 1.1

Nom	Utilisation	Rôle	Résultat
=	valeur1 = valeur2	Égalité	VRAI si les deux valeurs testées sont égales
≠	valeur1≠ valeur2	Inégalité	VRAI si les deux valeurs testées sont différentes
>	valeur1 > valeur2	Supérieur strictement	VRAI si valeur1 strictement supérieure à valeur2
<	valeur1 < valeur2	Inférieur strictement	VRAI si valeur1 strictement inférieure à valeur2
⩾	valeur1 ⩾ valeur2	Supérieur ou égal	VRAI si valeur1 supérieure ou égale à valeur2
⩽	valeur1 ⩽ valeur2	Inférieur ou égal	VRAI si valeur1 inférieure ou égale à valeur2

La syntaxe est la suivante :

```
SI (condition) ALORS
  instruction(s)
FINSI
```

Implantation C

```
if (condition)
{
  instruction(s);
}
```

Cette structure fonctionne de la manière suivante :

- si la condition est vraie, alors les instructions écrites entre les accolades sont exécutées ;
- si la condition est fausse alors, les instructions ne sont pas exécutées.

Structure SI...ALORS...SINON

Il arrive assez souvent qu'en fonction de la valeur d'une condition, le programme doive exécuter des instructions si elle est vraie et d'autres instructions si elle est fausse. Plutôt que de tester une condition puis son contraire, il est possible d'utiliser la structure `SI...ALORS...SINON`, dont la syntaxe est la suivante :

Langage algorithmique

```
SI condition ALORS
  instructions 1
SINON
  instructions 2
FINSI
```

Implantation C

```
if (condition)
{
  instructions 1;
}
else
{
  instructions 2;
}
```

La partie `SI...ALORS` est identique à la structure de contrôle simple : si la condition est vraie, alors les instructions 1 sont exécutées, et pas les instructions 2. Les instructions 2, concernées par le `SINON`, sont exécutées si et seulement si la condition est fausse.

Dans un programme, l'utilisation de tests est très fréquente. Quelle forme aurait un programme dans lequel il serait nécessaire de vérifier deux, quatre, voire dix conditions avant de pouvoir exécuter une instruction ? En vertu des règles d'indentation et de présentation, il deviendrait rapidement illisible et moins compréhensible. Afin d'éviter cela, il est possible de regrouper plusieurs conditions en une seule en utilisant d'autres opérateurs logiques (booléens) encore appelés **connecteurs logiques**.

Ces opérateurs permettent d'effectuer des calculs avec des valeurs de type VRAI ou FAUX et de fournir un résultat de type VRAI ou FAUX. Nous en présentons trois, nommés `ET`, `OU` et `NON` en indiquant simplement les résultats qu'ils produisent. Ces tableaux sont nommés **tables de vérité** :

- `ET` : intuitivement, la condition (c1 `ET` c2) est VRAI si et seulement si la condition c1 est VRAI ET si la condition c2 est VRAI.
- `OU` : la condition (c1 `OU` c2) est VRAI si et seulement si la condition c1 est VRAI OU si la condition c2 est VRAI.
- `NON` : l'opérateur `NON` change quant à lui la valeur de la condition qu'il précède. Il se note également ¬.

Tableau 1.2

Opérateur ET			Opérateur OU			Opérateur NON	
c1	**c2**	**c1 ET c2**	**c1**	**c2**	**c1 OU c2**	**c1**	**NON c1**
FAUX	FAUX	FAUX	FAUX	FAUX	FAUX	FAUX	VRAI
FAUX	VRAI	FAUX	FAUX	VRAI	VRAI		

Implantation C

```
ET  se note &&
OU  se note ||
¬   se note !
```

1.5.3 Itérations et boucles

Certains algorithmes nécessitent de répéter des instructions un certain nombre de fois avant d'obtenir le résultat voulu. Cette répétition est réalisée en utilisant une structure de contrôle de type itératif, nommée **boucle**. Il existe trois types de boucles.

La boucle TANT...QUE

Sa syntaxe est la suivante :

Langage algorithmique

```
TANTQUE condition FAIRE
  instruction 1
  ...
  instruction n
FAIT
instructions suivantes
```

Implantation C

```
while (condition)
{
  instruction 1;
  ...
  instruction n;
}
instructions suivantes;
```

Lorsque l'ordinateur rencontre cette structure, il procède systématiquement de la manière suivante :

- La condition est testée (on dit aussi évaluée).
- Si la condition est **fausse**, l'instruction ou les instructions du bloc ne sont pas exécutées et on passe aux instructions suivantes (après la structure de contrôle).
- Si la condition est **vraie**, l'instruction ou les instructions du bloc sont exécutées, et on recommence à l'étape 1) : test de la condition.

La boucle FAIRE...TANT QUE

Cette structure de contrôle est très proche syntaxiquement de la boucle ou répétition `TANTQUE`. La seule différence réside dans l'ordre dans lequel sont faits les tests et les instructions. Cette structure s'utilise de la manière suivante :

Langage algorithmique

```
FAIRE
  instruction 1
  ...
```

```
  instruction n
TANTQUE condition
instructions suivantes
```

Implantation C

```
do
{
  instruction 1;
  ...
  instruction n;
}
while (condition);
instructions suivantes;
```

Lorsque l'ordinateur rencontre cette structure, il procède systématiquement de la manière suivante :

- Exécution de l'instruction ou du bloc d'instruction concernée.
- Test de la condition (on dit aussi évaluation).
- Si la condition est **vraie**, l'instruction ou les instructions du bloc sont exécutées, et on recommence à l'étape 1 soit *exécution de l'instruction ou des instructions du bloc*.
- Si la condition est **fausse**, le programme passe aux instructions suivantes.

La boucle POUR

Lorsque le nombre de fois où un bloc d'instructions doit être exécuté est connu à l'avance, la boucle `POUR` est préférable aux boucles précédentes. L'usage principal de la boucle `POUR` est de faire la gestion d'un compteur (de type entier) qui évolue d'une valeur à une autre.

Langage algorithmique

```
POUR variable DE valeur1 A valeur2 FAIRE
  instruction 1
  ...
  instruction n
FAIT
instructions suivantes
```

Implantation C

```
for (variable = valeur1; variable <= valeur2; variable++)
{
  bloc d'instructions;
}
instructions suivantes;
```

Lorsque l'ordinateur rencontre cette structure, il procède systématiquement de la manière suivante :

- La variable, jouant le rôle de compteur, est initialisée à la valeur1.
- L'ordinateur teste si la variable est inférieure ou égale à la valeur2 :
  - si c'est le cas, l'instruction ou le bloc d'instruction est effectué, la variable jouant le rôle de compteur est augmentée de 1, et retour à l'étape 2, et non à l'étape 1 qui initialise la variable ;
  - si ce n'est pas le cas, l'instruction ou le bloc d'instruction n'est pas effectuée, et l'ordinateur passe aux `instructions suivantes.`

En réalité, la boucle `POUR` est équivalente à la boucle `TANTQUE`, mais les deux sont utilisables dans des cas distincts. Dans le cas où le nombre d'itérations n'est pas connu à l'avance, la boucle `TANTQUE` sera utilisée.

1.6 Tableaux

Un programme peut être amené à manipuler de nombreuses variables représentant des valeurs distinctes mais de même nature. Par exemple, un relevé de plusieurs températures en plusieurs endroits et à plusieurs dates nécessitera autant de valeurs entières que de températures à stocker. Il est difficilement envisageable de définir « manuellement » autant de variables que de valeurs à stocker. Les **tableaux**, en informatique, permettent de résoudre ce problème en proposant la création de plusieurs variables de même type, d'une manière très compacte.

1.6.1 Définition

Un tableau se définit en indiquant son nom, le type des éléments stockés dans le tableau, ainsi que leur nombre, écrit entre crochets. Ce nombre se nomme également la **taille maximale** du tableau.

Syntaxe : `VAR nom_du_tableau : type_des_éléments[taille_maximale]`

> ***Exemple***
>
> `VAR tab : entier[100]` est la définition d'un tableau nommé tab qui peut stocker 100 valeurs de type entier au maximum.

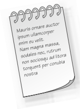


La taille maximale d'un tableau **doit** être une constante numérique.

Ce que n'est pas un tableau

Un tableau est en réalité une variable comme les autres, mais son type est d'une nature radicalement différent des types présentés plus haut. En particulier, ce n'est pas le type des éléments stockés. Ainsi :

- un tableau stockant des `entier` n'est pas un `entier` ;
- un tableau stockant des `reel` n'est pas un `reel` ;
- un tableau stockant des `caractere` n'est pas un `caractere`.



Il n'est pas possible de manipuler un tableau en utilisant les opérateurs arithmétiques et logiques que nous avons rencontrés jusqu'à présent, ceux-ci étant limités aux types de base. Il est très important de se rappeler cela, c'est un bon moyen d'éviter les erreurs lors de l'écriture d'un programme.

1.6.2 Représentation

Un tableau peut être vu comme un ensemble de « cases » où chaque case stocke une valeur. Soit la définition suivante :

```
VAR vals : réel[15]
```

Cette définition crée un tableau nommé `vals` qui stocke au maximum quinze `réel` dans quinze cases différentes. Aucune de ces cases n'est initialisée, ce qui est indiqué par un « ? ».

Tableau 1.3 **vals**

?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Chacune de ces cases est repérée par son numéro ou *indice* à l'intérieur du tableau. Un tableau de taille maximale *N* verra ses cases numérotées de 0 à *N*–1. En effet, pour un ordinateur, la valeur 0 est une valeur comme une autre.

Soit *i* un indice (*i* est donc de type entier puisqu'il s'agit d'un numéro de case). La valeur stockée dans la case d'indice *i* d'un tableau tab se nomme `tab[i]`.

Tableau 1.4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
tab[0]		tab[2]				tab[6]								tab[14]

Remarque

Cette schématisation sera employée à de nombreuses reprises dans cet ouvrage. N'hésitez pas à l'utiliser lors de la résolution des exercices, elle aide à visualiser le déroulement des algorithmes.

Il ne faut pas confondre la valeur notée entre crochets lors de la définition du tableau (la taille maximale) et la valeur notée entre crochets lors des instructions (l'indice).

Toute expression qui donne une valeur entière peut jouer le rôle d'indice lors de l'accès aux valeurs stockées dans un tableau.

La notation `t[i]`, où `t` est un tableau et `i` un indice est équivalente à une variable et peut être utilisée comme telle dans un programme. Cette variable est considérée comme ayant le type des éléments stockés dans le tableau.

Soient les définitions de variables suivantes :

```
VAR  x : réel
VAR  z : réel[20] // z est un tableau stockant au plus 20 réels
VAR  idx : entier // variable utilisée comme un indice
```

Les instructions suivantes sont alors valables :

```
x      ← 1.205E-17
z[2] ← x // car z[2] et x sont tous deux des réels
idx   ← 4
z[idx] ← x + z[idx-2] // soit z[4] ← x + z[2]
```

Taille utile

Le fait de donner la taille maximale d'un tableau sous la forme d'une constante est une contrainte lourde, puisque cette valeur est fixée lors de l'écriture du programme. Or un tableau ne stockera pas, en pratique, toujours autant d'éléments que sa taille maximale. Il est donc nécessaire de savoir combien de variables sont réellement intéressantes à traiter. Ce nombre de variables doit être stocké dans une variable de type entier nommé **taille utile** du tableau par opposition à la taille **maximale** fournie à la définition du tableau. Dans un tableau dont la taille **utile** est P et dont la taille **maximale** est N, les variables intéressantes seront rangées aux indices compris entre 0 et P–1. Les cases dont les indices vont de P à N–1 stockent des valeurs (car une variable n'est jamais vide) mais elles ne seront pas concernées par les traitements ou instructions effectuées.

À chaque définition d'un tableau est associée la définition de sa taille utile. Ceci doit être systématique.

Cette taille utile peut évoluer lorsque :

- Un élément est ajouté au tableau, dans ce cas la taille utile est augmentée de 1 (ou encore incrémentée). Avant d'ajouter un élément, il faut vérifier qu'il reste au moins une case disponible : la taille utile doit être inférieure strictement à la taille maximale du tableau. Cette vérification est réalisée par l'emploi d'un test (instruction si).
- Un élément est retiré du tableau, dans ce cas la taille utile est diminuée de 1 (ou décrémentée). Avant d'enlever un élément du tableau, il faut vérifier que la taille utile est strictement positive, c'est-à-dire qu'il y a au moins un élément dans le tableau.

Enfin, cette variable stockant la taille utile d'un tableau n'a pas de nom qui lui est spécifiquement dédié. Pour des raisons de lisibilité, cette variable sera nommée `util` ou `tai_ut` par exemple.

1.6.3 Relation entre tableaux et boucles

Les boucles sont extrêmement utiles pour les algorithmes associés aux tableaux. En effet, de nombreux algorithmes relatifs au tableau nécessitent de parcourir les éléments du tableau dans un certain ordre, le plus souvent dans le sens des indices croissant. Le traitement de chacun des éléments étant souvent le même, seule la valeur de l'indice est amenée à changer. Une boucle est donc parfaitement adaptée à ce genre de traitements.

Illustration par un algorithme de recherche de la plus grande valeur stockée dans un tableau d'entiers. Cet algorithme est abondamment commenté car il est une synthèse des notions rencontrées pour les tableaux.

```
PROGRAMME recherche_max
VAR maxi :entier            // stocke la valeur du maximum
VAR tabloval :entier [50]   // un tableau stockant des valeurs
VAR t_ut : entier           // la taille utile du tableau
VAR cpt : entier            // index des éléments du tableau
VAR rep : caractere         // pour la saisie des valeurs
// étape numéro 1 : initialisation des variables
t_ut ← 0  // à ce stade, pas de variable dans le tableau
FAIRE
  AFFICHER("entrez une valeur dans le tableau : ")
           // valeur rangée dans la case d'indice t_ut
  SAISIR(tabloval[t_ut])
           // incrémentation de t_ut (car ajout d'un élément)
  t_ut ← t_ut+1 ;
  AFFICHER("une autre saisie ? (o/n) :")
  SAISIR(rep);
  // la boucle reprend s'il reste de la place
  // dans le tableau ET si l'utilisateur souhaite continuer
TANTQUE ((t_ut < 50) ET (rep='o'))
  // pour l'instant, le plus grand est dans la case 0
maxi ← tabloval[0]
           // cherchons case par case (de l'indice 1 à t_ut-1)
POUR cpt DE 1 A t_ut-1 FAIRE
  // si l'on trouve plus grand :
  SI (tabloval[cpt] > maxi) ALORS
          // la valeur est mémorisée dans maxi
      maxi ← tabloval[cpt]
  FINSI
FAIT
```

1.6.4 Les tableaux à plusieurs dimensions

Il est possible de définir des tableaux à plusieurs dimensions en les indiquant dans des crochets successifs lors de la définition du tableau. Pour des propos d'illustration l'exemple se limitera à deux dimensions, la généralisation à N dimensions est immédiate.

Certains problèmes (notamment les jeux de plateau) ont une représentation naturelle en deux dimensions avec un repérage en lignes/colonnes ou abscisse/ordonnée.

Exemple

Un damier se représente comme un plateau de 100 cases constitué de 10 lignes et 10 colonnes. Une programmation d'un jeu de dames utilisera donc de manière naturelle un tableau à deux dimensions, une pour les lignes, l'autre pour les colonnes. L'état de chacune des cases du damier sera stocké sous la forme d'un entier (1 pour vide, 2 pour pion blanc, etc.). Ainsi la définition

> d'un tableau dans ce cadre sera la suivante :
> // 10 lignes, chaque ligne ayant 10 colonnes, soit 100 cases entier damier[10][10]

Cette définition permet de simplifier la représentation et donc la résolution du problème !

Utilisation d'indices dans les tableaux à deux dimensions : chaque élément du tableau est repéré par un numéro de ligne et un numéro de colonne. Ainsi, si `lig` et `col` sont deux indices (donc des entiers) valides (compris entre 0 et 9 pour l'exemple du damier), `damier[lig][col]` est l'entier situé à la ligne `lig` et à la colonne `col` du tableau à deux dimensions pris en exemple.



Attention tout de même à cet exemple : les notions de *ligne* et *colonne* ne sont pas connues par l'ordinateur, qui ignore ce qui est fait des valeurs qu'il stocke.

1.7 Pointeurs

Abordons maintenant un point souvent redouté à tort : les **pointeurs**. Un peu de logique et de rigueur suffisent à bien comprendre cette notion fondamentale. Une fois de plus, la notion de type sera essentielle dans cette partie.

1.7.1 Notion d'adresse

Toute variable possède trois caractéristiques : un **nom** et un **type**, qui ne peuvent être modifiés au cours du programme, car fixés au moment de la définition de la variable ; et une **valeur** qui au contraire évolue au cours du programme. En réalité, toute variable possède également une autre caractéristique fondamentale, utilisée en interne par la machine : une **adresse**. En effet, afin de mémoriser la valeur d'une variable, l'ordinateur doit la stocker au sein de sa mémoire, mémoire dont les éléments ou cellules sont repérés par des numéros (de manière analogue à ce qui se passe dans un tableau en réalité). Le nom de la variable n'est en fait pas utile à la machine, qui se contente de son adresse ; c'est pour le confort du programmeur que le choix du nom est rendu possible. Ainsi, il existe une dualité entre le nom de la variable (côté programmeur) et son adresse (côté machine).

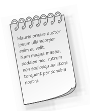
L'adresse d'une variable n'est autre que le numéro de la case ou cellule mémoire que celle-ci occupe au sein de la machine. Au même titre que son nom, l'adresse d'une variable ne peut pas varier au cours d'un programme. Il n'est même pas possible de choisir l'adresse d'une variable, cette adresse est attribuée automatiquement par l'ordinateur.

Chose curieuse, ces adresses de variables seront manipulées sans même connaître leur valeur exacte. Il suffira de savoir les nommer pour les utiliser.

Pour ce faire, un nouvel opérateur est nécessaire : l'opérateur `&`. Il se lit tout simplement « adresse de » et précède uniquement un nom de variable.

> ***Exemple***
>
> Soit la définition de variable suivante : **VAR** x : réel
> Cette simple définition attribue à la variable : un nom (x), un type (réel), une valeur (inconnue), et également une adresse (de manière automatique).

La notation &x signifie donc : « adresse de x ». La valeur précise de cette adresse ne nous est en réalité d'aucun intérêt, mais il sera parfois nécessaire de la manipuler.



Toute variable possède une et une seule adresse.

1.7.2 Définition et contenu

Un **pointeur** est une variable un peu particulière, car elle ne stocke ni un entier, ni un réel, ni un caractère, mais une adresse. Il est tentant de penser qu'une adresse étant un numéro de cellule dans la mémoire, elle est comparable à un entier. Cependant, une adresse ne s'utilise pas comme un entier, puisqu'une adresse ne peut pas être négative, et ne sert pas à faire des calculs. Une adresse est donc en ce sens un nouveau type.

Notion de contenu

Soit p un pointeur (il sera temps de passer à la syntaxe exacte de définition de pointeur lorsque toutes les notions nécessaires auront été abordées). p étant un pointeur, il est également une variable, et donc possède un nom, une valeur (qui est une adresse) et un type.

Un pointeur possède en plus une autre caractéristique, qui est son contenu.

Une adresse est le numéro d'une cellule mémoire, et dans cette cellule, se trouve une valeur. Cette valeur est appelée le contenu du pointeur, et ne doit pas être confondue avec sa valeur.

Le contenu d'un pointeur est la valeur de la cellule mémoire dont ce pointeur stocke l'adresse.

Puisque ces deux notions sont différentes, il existe une nouvelle notation pour indiquer l'accès à un contenu : cette notion est l'opérateur * (lire étoile). Cet opérateur est le même que l'opérateur de multiplication, mais le contexte d'écriture permet à l'ordinateur de reconnaître quelle est la signification exacte de cet opérateur.

Voici une règle très simple et très utile pour l'utilisation de cet opérateur *: il se lit toujours comme « contenu de », et non pas « pointeur ». Cette règle évitera de nombreuses confusions par la suite.

Exemple

Soit ptr un pointeur. Supposons que la valeur de ce pointeur soit 25040 (rappelons que cette valeur est tout à fait arbitraire). Supposons également que dans la mémoire, à l'emplacement 25040 se trouve la valeur 17.
Dans ce cas, la valeur de ptr, notée ptr, est 25040.
Le contenu de ptr, noté *ptr, est la valeur de la cellule mémoire numéro 25040, c'est-à-dire 17.
*ptr vaut donc 17

Type pointé

Quelle est la syntaxe permettant de définir un pointeur ? Cette syntaxe n'est hélas pas immédiate (sinon elle aurait déjà été présentée), car le type « pointeur » n'existe pas en tant que tel. Il

est nécessaire d'ajouter des informations supplémentaires afin que l'ordinateur puisse exploiter correctement ces pointeurs.

En réalité, un pointeur a besoin d'informations sur son contenu, sinon il ne peut rien en faire. La valeur d'un pointeur n'est qu'une adresse, mais cette information n'est pas suffisante pour gérer les contenus possibles.

Les différents types déjà évoqués auparavant, `entier`, `reel` et `caractere`, ont des tailles différentes : un entier occupe quatre octets (ou cellules), un réel en occupe huit et un caractère un seul. Lorsque l'ordinateur cherche à accéder au contenu d'un pointeur en mémoire, il doit savoir combien de cellules seront concernées par cette opération. Le type de contenu, encore appelé **type pointé**, est indispensable à la définition d'un pointeur. Il lui est même tellement lié qu'un pointeur ne peut pointer qu'un seul type de contenu.

Il n'existe donc pas de pointeur « générique » vers n'importe quel type de contenu, mais des pointeurs sur (ou vers) des `entier`, des pointeurs sur des `reel`, des pointeurs vers des `caractere`.

Un pointeur se définit par le type de son contenu. Pour la création d'un pointeur nommé ptr qui pointe sur des entiers, sa définition s'écrit : « le contenu de `ptr` est de type `entier` ». En langage informatique, c'est cette notation qui sera traduite en définition de variable, de la manière suivante : `VAR *ptr : entier` (se lisant « le contenu de `ptr` est de type `entier` »). Puisque le contenu de `ptr` est de type entier, il est aisé d'en déduire que `ptr` est un pointeur vers un entier. Attention à ce point qui ne semble pas très important, mais qui est fondamental.

Il en est de même pour les autres types pointés : la définition d'un pointeur nommé `ptr_reel` vers un réel est la suivante : `VAR *ptr_reel : réel` (se lisant : « le contenu de `ptr_reel` est de type `reel` »). Enfin, la définition d'un pointeur nommé `p_cgs` vers un caractère est la suivante : `VAR *p_cgs : caractere` (se lisant : « le contenu de `p_cgs` est de type `caractere` »).

Dans les exemples de définition de pointeurs précédents, la convention de nommage d'un pointeur veut que son nom débute par `ptr` ou `p_`. Cependant un pointeur étant une variable, le choix de son nom n'est contraint que par les règles de nommage de variables exposées en début de chapitre.

1.7.3 Initialisation

Les dangers de l'opérateur « * »

Comme pour toute variable, un pointeur doit être **initialisé** avant d'être manipulé. Comme un pointeur stocke une adresse, il ne peut être initialisé comme une simple variable de type `entier`, `reel` ou `caractere` car les adresses sont gérées par la machine et non par l'utilisateur. Accéder à un contenu est une opération délicate car elle est en relation avec la mémoire, qui est une ressource indispensable au fonctionnement de l'ordinateur. Chaque programme y stocke les valeurs dont il a besoin pour s'exécuter correctement. Étant donné qu'un ordinateur de type PC classique fait cohabiter plusieurs dizaines de programmes différents au même instant, il doit déléguer la gestion de la mémoire à un programme particulier nommé le système d'exploitation. Le rôle de ce dernier est de veiller au bon fonctionnement de la partie *hardware* de l'ordinateur, partie qui concerne la mémoire.

En imaginant qu'un pointeur puisse stocker n'importe quelle valeur, accéder au contenu se révélerait particulièrement dangereux pour la stabilité du système. Un simple exemple devrait vous en convaincre.

Exemple

Soient la définition et l'initialisation de pointeur suivantes :

```
VAR *ptr : entier    // lire : "le contenu de ptr est de type entier"
ptr ← 98120   // initialisation de la valeur du pointeur
              // (ptr donne accès à la valeur de ptr)
*ptr ← -74    // initialisation du contenu du pointeur ptr
              // (*ptr est le contenu de ptr)
```

Ces simples instructions permettraient au programme d'écrire une valeur arbitraire à n'importe quelle adresse de la mémoire de l'ordinateur, ce qui n'est pas du goût du système d'exploitation. Dans le cas d'un accès à une adresse non autorisée, ce dernier met une fin brutale à l'exécution du programme responsable de cet accès : c'est une des sources des « plantages » des programmes, et même la source la plus fréquente de ce phénomène.
Afin d'éviter un trop grand nombre d'erreurs de ce type, l'ordinateur (le compilateur en réalité), refusera d'initialiser un pointeur avec des valeurs arbitraires.



Le seul cas où un pointeur stocke une valeur arbitraire est lorsqu'il vient d'être défini : comme toute autre variable, il stocke alors une valeur aléatoire. La règle d'or d'utilisation de l'opérateur * (accès au contenu, ou encore prise de contenu) est la suivante :
l'accès au contenu d'un pointeur ne se fait en toute sécurité que si ce dernier a été correctement initialisé.

La valeur NULL

Il est possible d'initialiser un pointeur avec une adresse qui est forcément inaccessible, quel que soit le programme écrit. L'utilité d'une telle initialisation est qu'elle permet de savoir par un simple test si le pointeur a été initialisé avec une adresse valide, et ainsi éviter les accès malencontreux à des adresses invalides. L'adresse qui est forcément inaccessible est l'adresse 0. Selon le type de machines, cette adresse stocke des informations plus ou moins vitales pour le compte du système d'exploitation. En informatique, l'adresse 0 porte un nom particulier : `NULL`, qui n'est qu'un autre nom qui est donné à la valeur 0, mais `NULL` ne s'utilise que dans des contextes particuliers. De nombreuses instructions utilisent d'ailleurs cette valeur notée `NULL`.

Exemple d'utilisation de NULL

```
PROGRAMME test_NULL
// lire "le contenu de p_val est de type réel" :
VAR *p_val : réel
// initialisation de p_val avec NULL
// que l'on sait inaccessible
p_val ← NULL
// plus loin dans le programme...
SI (p_val ≠ NULL)
```

```
  // instructions dans lesquelles on peut accéder à *p_val
SINON
  AFFICHER("attention, p_val vaut NULL, on ne peut accéder à *p_val")
FINSI
```

Dans ce cas, le programme affiche un message d'avertissement plutôt que de provoquer une erreur qui n'est pas évidente à repérer et à corriger : ce type de programmation est à privilégier dans la mesure où le programmeur maîtrise beaucoup mieux ce qui se passe au sein du programme.

Les adresses de variables existantes

Les variables que l'utilisateur définit au sein de son propre programme ont forcément des adresses valides. Pour rappel, l'adresse d'une variable est accessible au moyen de l'opérateur `&`.

Exemple

Voici donc un exemple d'initialisation de pointeur tout à fait valide :

```
VAR *ptr_c : caractère
VAR lettre : caractère
ptr_c ← &lettre
```

Les types sont compatibles, car `ptr_c` pointe vers un caractère, autrement dit stocke l'adresse d'une valeur de type caractère, et `&lettre` est l'adresse d'une variable de type caractère. Que se passe-t-il exactement dans ce cas ? Continuons le programme pour illustrer les relations entre les variables `ptr_c` et `lettre`.

```
lettre ← 'Y'
AFFICHER((*ptr_c)+1) // cette instruction affiche 'Z'
```

Explication avec des valeurs numériques

Lors de la définition de la variable `lettre`, une adresse lui est automatiquement attribuée par l'ordinateur. La valeur choisie est par exemple, 102628 (ce nombre n'a aucune importance, mais aide à illustrer l'explication). L'adresse de `lettre (&lettre)` vaut donc 102628, et `lettre` vaut `'Y'` suite à son initialisation par l'instruction `lettre ← 'Y'`.

Suite à l'exécution de l'instruction `ptr_c ← &lettre`, le pointeur `ptr_c` reçoit 102628 : la valeur de `ptr_c` est donc 102628.

Que vaut le contenu de `ptr_c (*ptr_c)` ? Le contenu du pointeur `ptr_c` est la valeur stockée en mémoire à l'adresse 102628 (car `ptr_c` vaut 102628). Il se trouve que cette adresse est celle de la variable lettre. Ainsi `* ptr_c` vaut `'Y'`. Il n'est donc pas étonnant que `(*ptr_c)+1` soit égal à `'Z'`.

Allocation dynamique de mémoire et commande RESERVER()

Le dernier type (et le plus intéressant) d'initialisation de pointeur consiste à effectuer cette initialisation à l'aide d'une nouvelle adresse autorisée. C'est rendu possible par l'emploi d'une nouvelle commande dont le but est d'obtenir une zone de stockage mémoire dynamiquement, c'est-à-dire lors de l'exécution du programme.

Cette commande se nomme `RESERVER()` et sa syntaxe d'utilisation est la suivante :

```
RESERVER(pointeur)
```

Le rôle de cette commande est d'allouer un espace mémoire pour stocker le contenu qui sera pointé par le pointeur qui est fourni à la commande.

> ***Exemple :***
>
> Soit le pointeur `ptr` défini de la manière suivante :
> `VAR *ptr : réel`
> A priori, `ptr` n'est pas initialisé et il est donc hasardeux d'accéder à son contenu. Afin d'initialiser ptr correctement, il est possible de reserver de l'espace mémoire pour son contenu.
> `RESERVER(ptr)` aura cet effet.

Une **réservation** de mémoire se nomme également une **allocation dynamique** de mémoire. Lorsque la place en mémoire n'est plus utile, il suffit de la libérer en utilisant la commande suivante : `LIBERER(pointeur)`

1.8 Les sous-programmes ou fonctions

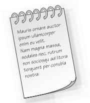
Le rôle d'une **fonction** est de regrouper des instructions ou traitements qui doivent être faits de manière répétitive au sein d'un programme. Par exemple, dans un programme traitant des tableaux, on voudrait afficher plusieurs fois des tableaux dont les variables stockent des valeurs différentes. Cependant, même si les valeurs sont différentes, l'affichage d'un tableau se résume toujours à un parcours de toutes les variables utilisées avec une boucle `POUR` (de l'indice `0` jusqu'à l'indice *`taille_utile-1`*), avec un affichage de chaque valeur grâce à l'instruction `AFFICHER()`.

Il serait utile de regrouper ces instructions d'affichage, pour n'en avoir qu'un seul exemplaire, et de pouvoir s'en servir lorsqu'on en a besoin, sans avoir à saisir les lignes dans le programme. C'est à cela que sert une fonction : elle regroupe des instructions auxquelles on peut faire appel, c'est-à-dire utiliser en cas de besoin. Il s'agit en réalité d'un petit programme qui sera utilisé par le programme : on parle aussi de **sous-programme** pour une fonction. Ce sous-programme fonctionne en « boîte noire » vis-à-vis des autres sous-programmes, c'est-à-dire qu'ils n'en connaissent que les entrées (valeurs qui sont fournies à la fonction et sur lesquelles son traitement va porter) et la sortie (valeur fournie par la fonction, qui peut être récupérée par les autres fonctions ou par le programme).

Une analogie avec une fonction mathématique permet d'illustrer clairement les notions d'entrées et de sorties. Soit la fonction suivante définie avec le formalisme des mathématiques :

$$F : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$$
$$x, y \rightarrow x^2 + xy + y^2$$

Les entrées sont les données que l'on fournit à la fonction pour qu'elle puisse les traiter et fournir un résultat : il s'agit des valeurs x et y. La sortie est le résultat que donne la fonction, il s'agit de la valeur calculée $x^2 + xy + y^2$.



Types des entrées et sorties – D'après la définition (mathématique) de la fonction, le type informatique des entrées et de la sortie peut être déterminé : ce sont des réels dans cet exemple.

La fonction proposée en exemple a comme entrées deux valeurs de type réel, effectue un calcul, et a comme sortie une valeur de type réel.

1.8.1 Définition d'une fonction

Comme de nombreuses entités en informatique, une fonction doit être définie avant d'être utilisée, c'est-à-dire que l'on doit indiquer quelles sont les instructions qui la composent : il s'agit de la définition de la fonction, où l'on associe les instructions à l'identification de la fonction.

On doit donc trouver une définition de la fonction, qui comporte :

- Une identification ou **en-tête de la fonction**, suivie des instructions de la fonction, ou **corps de la fonction**.



La fonction doit être définie et comporter : un en-tête, pour l'identifier et un corps contenant ses instructions, pour la définir.

Comment écrire une définition de fonction

- *En-tête*

L'en-tête d'une fonction contient les informations nécessaires pour identifier la fonction : son nom, ses entrées et **leur** type, le type de sa sortie (sans la nommer, car c'est inutile).

Syntaxe de l'en-tête

```
FONCTION nom(liste entrées avec leurs types ): type de la sortie
```

La liste des entrées avec leur type suit exactement la même syntaxe que les définitions de variables, et ceci n'est pas un simple hasard, car les entrées sont des variables de la fonction.

La seule différence réside dans le fait que si plusieurs entrées ont le même type, on ne peut pas les écrire comme on le ferait pour une définition multiple de plusieurs variables du même type. Il faut rappeler le type de l'entrée pour chacune des entrées.

La sortie ne nécessite pas d'être nommée, car c'est à la fonction ou au programme appelant la fonction de récupérer cette valeur. Le rôle de la fonction se limite uniquement, dans le cas de la sortie, à pouvoir fournir une valeur numérique, seul son type est donc important.

- *Corps*

Le corps de la fonction est constitué des instructions de la fonction, et est placé directement à la suite de l'en-tête de la fonction, entre les mots `DEBUT` et `FIN`.

Dans le corps de la fonction, outre les instructions, on peut également trouver des définitions de variables qui peuvent être utiles pour faire des calculs intermédiaires lorsque l'on utilise la fonction. Les définitions des variables se trouvent avant le mot `DEBUT`.

- *Les variables locales*

Ces variables, définies à l'intérieur de la fonction, sont des **variables dites locales**, car elles ne sont connues que par la fonction. C'est logique, car c'est la fonction qui les définit (en quelque

sorte) pour son usage propre. Une autre fonction ou un programme extérieur ne connaissent pas l'existence de ces variables, et donc à plus forte raison ne connaissent ni leur nom, ni leur type, ni leur valeur, ni leur adresse. Elles ne peuvent les utiliser.

Les entrées de la fonction, précisées dans l'en-tête, sont également considérées comme des variables locales de la fonction : c'est pour cela que la syntaxe d'écriture des entrées est si proche de la syntaxe d'une définition de variable.

Enfin, les définitions des variables locales à la fonction suivent exactement les mêmes règles que celles que nous avons déjà rencontrées pour les définitions de variables d'un programme.

- *L'instruction de retour*

Pour traiter intégralement les fonctions, nous avons besoin d'une nouvelle instruction permettant que la fonction communique sa sortie à la fonction qui l'utilise. La sortie d'une fonction (il y en a au maximum une) est en fait une valeur numérique que celle-ci fournit, et la seule information nécessaire à la bonne interprétation de cette valeur numérique est son type : c'est pourquoi on ne précise que le type de cette sortie, il est inutile de lui associer un nom. Il nous faudrait donc une instruction dont l'effet serait : « *répondre à la fonction appelante la valeur qu'elle demande* ». Cette instruction existe, son nom est `RETOURNER`.

> ***Exemple***
>
> **Syntaxe : RETOURNER** `expression`
> **Effet :** transmet la valeur de l'expression et met fin à l'exécution de la fonction.

Toute instruction placée après l'instruction `RETOURNER` est purement et simplement ignorée, ce qui fait que cette instruction s'emploie forcément comme dernière instruction d'une fonction.

La seule contrainte à respecter est que le type de l'expression soit le même que celui de la sortie, qui est indiqué dans l'en-tête de la fonction.

Lorsqu'une fonction ne possède pas de sortie, il n'est pas indispensable d'utiliser l'instruction retourner, à part pour indiquer que la fonction se termine, ce qui peut aussi être repéré par le mot `FIN` de la fonction.

1.8.2 Appel des fonctions

L'**appel d'une fonction** correspond à une demande de son utilisation, ceci est fait dans une autre fonction, dont par exemple le programme principal. Afin d'appeler une fonction, on doit préciser : son nom, ainsi que les valeurs que l'on fournit pour les entrées. On ne précise pas la valeur de la sortie, car c'est la fonction appelée qui est en charge de la fournir !

La fonction (ou programme principal) qui appelle (ou utilise) une fonction est dite : **fonction appelante**; la fonction qui est utilisée est dite **fonction appelée**.

> ***Exemple***
>
> Soit une fonction nommée `fonc`, et possédant une liste d'entrées : `type1 entree1, type2 entree2,... ,typen entreen`. Son en-tête est donc :
>
> ```
> FONCTION fonc(entree1 : type1, entree2 : type2,..., entreen : typen):
> type_sortie)
> ```

Pour appeler cette fonction, la syntaxe est la suivante :

```
fonc(expr1, expr2,...,exprn)
```

où fonc est le nom de la fonction, et expr*i*, où *i* est compris entre 1 et *n*, est une expression dont le type est celui de l'entrée *i*.

Gestion des entrées : les arguments

Lors de l'appel de la fonction, une expression donnant une valeur à une entrée de la fonction est appelée **argument de la fonction**.



Retenez bien qu'un argument n'est pas forcément une variable, mais peut être une expression.

Dans tous ces cas, l'ordinateur réalise les deux étapes suivantes :

- Calcul de la valeur de l'expression pour chacun des arguments.
- Transmission de cette valeur à l'entrée de la fonction correspondante pour que la fonction s'exécute avec les valeurs transmises.

Gestion de la valeur de sortie de la fonction : affectation

Lorsqu'une fonction est appelée de cette manière, la valeur de la sortie retournée par la fonction appelée est obtenue ainsi : la valeur numérique de l'expression suivant l'instruction `RETOURNER` est calculée puis transmise à la fonction appelante. Au sein de cette fonction appelante, l'écriture `nom_de_fonction(liste_des_arguments)` est en réalité une expression dont le type est celui de la sortie de la fonction, qui est précisé dans l'en-tête de cette dernière. Il s'agit donc d'une valeur numérique que l'on doit affecter à une variable si on veut la conserver.

Lorsque vous voulez conserver le résultat retourné par une fonction appelée, il faut affecter ce résultat (obtenu par un appel) dans une variable du même type que celui de la sortie de la fonction appelée.

Le passage des paramètres

Nous allons étudier, dans cette partie, le mécanisme de passage des paramètres, qui est un point fondamental concernant les fonctions. Nous allons voir en détail la manière dont se fait la communication entre les arguments fournis par la fonction appelante et les paramètres (ou entrées) reçus par la fonction appelée. Ce passage de paramètre est effectué à chaque appel d'une fonction.

Le mécanisme de recopie

Les actions effectuées par l'ordinateur lors du passage des paramètres (transmission de la valeur des arguments au moment d'un appel de fonction) sont les suivantes :

- les valeurs des arguments sont **calculées** (ou évaluées) ;

- ces valeurs sont **recopiées** dans les paramètres correspondants de la fonction : l'ordre de recopie est celui dans lequel les entrées ou paramètres de la fonction sont écrits dans l'en-tête de la fonction : l'argument 1 dans le paramètre 1, et ainsi de suite ;
- la fonction **est exécutée** et réalise ses calculs et instructions ;
- l'instruction `RETOURNER` **est exécutée** : l'expression contrôlée par cette instruction est évaluée et retournée au surprogramme qui a appelé cette fonction en tant que sous-programme.

Intégrité des variables locales

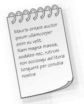
Les variables locales à un programme ou à une fonction ne peuvent pas être modifiées par une autre fonction lorsque l'on applique ce mécanisme pour des paramètres de type simple (`entier`, `caractere` ou `reel`). En effet, il ne faut pas confondre les arguments d'une fonction appelée et les variables du programme qui appelle cette fonction. Un argument n'est pas nécessairement une variable, et même si c'est le cas, c'est la valeur de l'argument qui est transmis à la fonction et non la variable elle-même. On peut donc en conclure qu'une variable utilisée comme argument d'un appel de fonction ne sera pas modifiée par l'appel, car c'est simplement sa valeur qui est recopiée.

1.8.3 Les fonctions et les tableaux

Syntaxe utilisée pour les entrées de type 'tableau'

Un tableau est une adresse, et c'est donc une adresse qui sera transmise par le biais d'une entrée d'un tel type. Il n'est pas possible de transmettre, avec un seul paramètre, d'autres informations que l'adresse, c'est-à-dire la taille utile ou même la taille maximum du tableau. Pour transmettre l'une de ces informations, il faudra utiliser un paramètre supplémentaire. Ainsi, un paramètre de type tableau sera défini comme un tableau contenant un certain type de valeurs, mais sans fournir ni la taille maximum, ni la taille utile.

> ***Exemple***
>
> Une entrée de type tableau sera fournie de la manière suivante :
> `nom_entree : type_des_valeurs_stockées[]`
> Les crochets ne contiennent aucune valeur, ils sont juste présents pour indiquer que le type de l'entrée est un tableau contenant des valeurs d'un certain type.
> Lorsque l'on voudra traiter les valeurs stockées dans ce tableau, il faudra par contre avoir une information concernant la taille utile de ce tableau : il faudra donc associer systématiquement cette entrée à une entrée de type tableau.



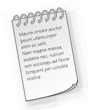
N'oubliez pas que, même si la taille utile est définie dans le programme principal ou dans une autre fonction que celle qui traite le tableau, cette taille utile sera stockée dans une variable à laquelle la fonction ne pourra pas accéder ! Il faudra donc la lui transmettre.

Appel d'une fonction avec un argument de type tableau

Lorsqu'un tableau est utilisé comme un argument, il est inutile d'utiliser la notation avec les crochets. En effet, ces crochets sont utilisés pour définir le tableau, ou pour accéder à un élément

particulier stocké dans le tableau. Le tableau lui-même (c'est-à-dire l'adresse à laquelle sont stockées les valeurs), est repéré par son nom, sans les crochets.

> ***Exemple***
>
> Soit la définition suivante :
> `tab_car : caractere[20]`
> cela indique que tab_car est un tableau contenant au plus 20 caractères. Donc tab_car est de type : tableau de caractere, ou encore pointeur vers des caractères. tab_car est une adresse.



Lorsque l'on veut fournir à une fonction un argument qui est un tableau, on doit lui fournir une adresse.

1.8.4 Les fonctions et les pointeurs

Le passage de tableau en paramètre est en fait une très bonne illustration des effets de la transmission d'une adresse à une fonction : le même phénomène entre en jeu lorsque ce sont des pointeurs qui sont utilisés pour ce passage de paramètres, puisqu'un tableau est un pointeur.

Nous avons ainsi remarqué que le passage d'une adresse, dans le cas d'un tableau, permet d'obtenir un accès à une variable du programme principal à partir d'une fonction recevant l'adresse de cette variable. Nous allons donc utiliser explicitement cette transmission d'adresse de variable pour avoir un accès à son contenu, et ce grâce à la notation `'*'` déjà traitée dans la section concernant les pointeurs.

Paramètre de type « adresse de »

La syntaxe utilisée pour un paramètre de fonction lors de la définition de celle-ci, lorsque le paramètre est un pointeur est la suivante :

```
*nom_du_paramètre : type_pointé
```

Cela revient donc à utiliser la syntaxe de définition d'une variable de type pointeur. Pour un paramètre, cette écriture s'interprète un peu différemment, même si cette interprétation est tout à fait cohérente avec tous les aspects abordés avec les pointeurs.

> ***Exemple***
>
> Le paramètre `nom_du_paramètre` est l'adresse d'une valeur de `type_pointé`.
> Pourquoi faire cette distinction ici ? Tout simplement parce qu'un argument fourni à une fonction lors de son appel est une expression, et non une variable : cela signifie, entre autres, que lors de l'appel à une fonction dont un paramètre est un pointeur, l'argument associé ne devra pas obligatoirement être un pointeur, mais tout simplement une adresse.
> Il pourra donc s'agir : de l'adresse d'une variable existante ou d'une adresse stockée dans un pointeur.

Les données modifiées

Les paramètres d'une fonction sont une copie des arguments. Ainsi, la modification de la valeur d'un paramètre n'aura aucune influence sur la valeur de l'argument. Une fonction ne peut pas modifier la valeur des arguments qui lui sont transmis. Cependant, il est parfois intéressant qu'une fonction puisse modifier une variable de la fonction qui l'appelle. Pour ce faire, la technique consiste à transmettre à la fonction l'adresse de la variable à modifier. Ainsi, la fonction récupère l'adresse de la variable et accède à sa valeur grâce à l'opérateur * (contenu de). La notation « *données modifiées* » rencontrée dans la suite de cet ouvrage indique que c'est une adresse qui est transmise.

Retour d'une adresse

Une fonction peut retourner une adresse, car n'importe quel type peut être fourni en sortie de fonction. Les précautions à prendre lorsqu'une adresse est retournée par une fonction sont les suivantes, ce sont des règles de programmation à respecter de façon très stricte :

- une fonction ne doit jamais retourner l'adresse d'un de ses paramètres ;
- une fonction ne doit jamais retourner l'adresse d'une de ses variables locales ;
- une fonction ne doit jamais retourner un tableau statique local (défini dans la fonction) ;
- une fonction peut retourner une allocation obtenue par la fonction `RESERVER()`.

1.9 Création de types par le programmeur : les types composés ou structures

À l'instar des tableaux permettant de regrouper des variables de même type, il existe la possibilité de regrouper des variables de types quelconques au sein de types crées par le programmeur en fonction de ses besoins. En effet, les types de base que sont réel, entier, caractère et pointeur (adresses) sont souvent très limitatifs lorsque le programmeur souhaite développer des applications professionnelles. Sans entrer dans les détails de l'utilisation précise de ces nouveaux types, il est possible de dire que ces types sont issus d'un travail de modélisation, travail dont le but est de choisir, pour un objet du monde réel qui doit être manipulé dans une application informatique, la liste des propriétés (valeurs) qui vont représenter cet objet.

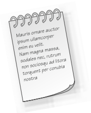
> ***Exemple***
>
> L'objet « personne » du monde réel possède énormément de propriétés (vous pourriez en dresser une liste très longue); mais toutes ne sont pas forcément adaptées pour une application précise. Un logiciel de gestion de compte bancaire n'aura par exemple pas besoin de connaître la taille, le poids, la couleur des cheveux ni le plat préféré d'une personne.

Pour définir un type composé, il faut en premier lieu lui choisir un nom, puis dresser la liste de toutes les propriétés, également appelées champs, que l'on souhaite stocker à l'intérieur de ce type. Chacun des **champs** est identifié par un nom, ce qui permet au programmeur de le choisir parmi ceux qui sont stockés dans le type composé, et d'un type, pour que l'ordinateur sache le manipuler.

Par convention de nommage, un nom de type composé doit systématiquement commencer par 't_'.

Pour définir un type composé, aussi appelé `structure`, on utilise simplement la syntaxe suivante :

```
STRUCTURE nom_du_type
    nom_champ_1 : type_champ_1
    ...
    nom_champ_n : type_champ_n
```



Le mot `STRUCTURE` permet de définir un nouveau type, et non pas une variable.

Définition du type 'complexe' pour représenter un nombre complexe :

```
STRUCTURE t_complexe
  // les définitions de champ sont comme
  // les définitions de variables, il est possible
  // de les regrouper.
  re : reel
  im : reel
```

Une fois ce type composé défini, il est utilisable dans un programme presque au même titre que les types de base de l'algorithmique, qui sont entier, réel et caractère. Il est notamment possible de définir des variables dont le type est un type composé ou structure, en utilisant la syntaxe classique de définition d'une variable :

```
VAR nom_de_variable : type
```

`VAR var_comp : t_complexe` est une définition de variable tout à fait correcte, et dont la signification est : la variable nommée `var_comp` est de type `t_complexe`.

1.9.1 Accès aux champs

L'opérateur utilisé pour accéder à un champ à partir du nom d'une variable est l'opérateur '.', dont la syntaxe d'utilisation est la suivante :

```
nom_de_variable.nom_du_champ
```

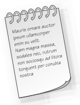
Cela s'interprète comme : le champ `nom_du_champ` de la variable `nom_de_variable`. Cette écriture est une expression dont le type est celui du champ référencé.

Exemple

Soit la définition de variable suivante : `VAR z : t_complexe`
Afin d'initialiser cette variable `z`, il est nécessaire d'accéder individuellement à ses champs, car l'ordinateur est incapable d'initialiser directement une variable de type `t_complexe`.

Pour stocker la valeur `1-2i` dans la variable `z`, il faut en réalité stocker la valeur `1` dans la partie réelle de `z` et `-2` dans la partie imaginaire de `z`, ce qui s'écrit :

```
z.re ← 1.0
z.im ← -2.0
```



Une confusion fréquente consiste à faire précéder le `'.'` du nom du type défini à l'aide de la structure. Le `'.'` doit systématiquement être précédé d'un nom de **variable**.

1.9.2 Opérateur d'affectation ←

L'opérateur d'affectation ← est le seul opérateur que l'ordinateur peut appliquer aux structures, car il s'agit dans ce cas de recopier les champs d'une variable de type composé dans les mêmes champs d'une autre variable du même type. C'est d'ailleurs tout ce que fait cette opération.

Une instruction d'affectation entre deux variables d'un type composé copie donc les champs d'une variable dans une autre.

1.9.3 Structures contenant des tableaux et des pointeurs

Tableaux statiques

Puisqu'un champ peut être de n'importe quel type, il peut s'agir entre autres d'un tableau, par exemple statique. Il est nécessaire, dans ce cas, de stocker sa taille utile dans un champ de la structure.

Lorsque plusieurs variables d'un tel type sont définies, un tableau statique différent (d'adresse différente) est créé pour chacune de ces variables. En cas d'affectation, ce sont les contenus des cases du tableau qui sont recopiés.

1.9.4 Structures définies à l'aide de structures

Un champ d'une structure peut être de n'importe quel type, donc il peut être d'un type composé, c'est-à-dire une structure. Cela aide à bien hiérarchiser les différents niveaux pour éviter que les structures ne comportent trop de champs.

Il est possible dans ce cas d'avoir un accès à un champ en utilisant plusieurs fois de suite la notation '.'.

Pour qu'un champ d'une structure `s1` soit lui-même une structure `s2`, il faut cependant que cette structure `s2` soit définie avant `s1`.

Exemple

```
STRUCTURE t_date
  jj, mm, aa : entier // pour jour, mois, année
STRUCTURE t_evenement
  ladate : t_date // réutilisation du type t_date défini précédemment
  *description : caractère // ou encore description : caractère[50]
```

le type t_evenement est défini à l'aide du type t_date.

1.9.5 Pointeurs vers les structures

Est-il possible d'utiliser des pointeurs pour stocker l'adresse d'une variable de type composé ? La réponse à cette question a des conséquences assez importantes, dans la mesure où, comme pour les exemples déjà traités dans les chapitres précédents, des applications (programmes professionnels utilisés dans l'entreprise) auront à utiliser beaucoup de structures.

Or, la possibilité de stocker en mémoire un certain nombre de structures implique que l'on puisse utiliser des tableaux, et le plus souvent, des tableaux dynamiques. En réalité, les tableaux statiques ne sont quasiment pas utilisés, seule la notation `[ ]` est vraiment confortable d'un point de vue de la programmation. Un petit retour en arrière vers les pointeurs est ici nécessaire.

Afin de pouvoir manipuler un contenu, l'ordinateur a besoin de deux informations :

- une **adresse** lui permettant de déterminer l'endroit de la mémoire à consulter ;
- un **type**.

Cette information de type, indispensable à la définition d'un pointeur, est exploitée pour savoir combien d'octets doit manipuler l'ordinateur lors de l'accès à la mémoire.

Les champs d'une variable de type composé sont stockés les uns à la suite des autres de manière consécutive dans la mémoire, cette variable forme un bloc en mémoire. Elle a donc une adresse (comme pour les tableaux, celle du début du bloc), ou plus précisément, une seule adresse permet de la repérer.

En ce qui concerne la taille de cette variable, le raisonnement est encore plus simple : la taille d'une variable de type structure est la somme de la taille de tous ses champs.

Ayant une taille fixe (et calculable par la méthode exposée ci-dessus) et une adresse unique, une variable de type `structure` peut être manipulée comme un contenu et donc il est possible de définir un pointeur vers une variable de type structure

Aspects syntaxiques

Les notions d'adresse, de valeur, de contenu restent dans ce cas tout à fait valides, et les notations usuelles s'appliquent. Soit *t_com* un type composé (une structure). Définir un pointeur *ptr* vers un contenu de type *t_com* s'écrit donc naturellement :

```
VAR *ptr : t_com
```

Pour initialiser ce pointeur ptr, les règles ne changent pas, il faut lui donner une adresse valide : celle d'une variable existante ou l'adresse d'une zone de mémoire obtenue par l'emploi de la réservation.

Accès aux champs

Comment accéder aux champs d'une variable de type structure lorsque l'on ne dispose que d'un pointeur vers cette variable et non de sa valeur ? Dans ce cas, la valeur de la variable n'est autre que le contenu du pointeur.

Exemple

Soit p_tot un pointeur défini de la manière suivante :
VAR *p_tot : t_complexe
Son initialisation est la suivante :
RESERVER(p_tot)
C'est donc le contenu de la zone mémoire pointée par `p_tot` et noté `*p_tot`, qui est de type `t_complexe` (regardez la définition du pointeur `p_tot`, elle ne dit pas autre chose). Il est alors possible d'accéder à ses champs comme pour n'importe quelle autre variable de ce type, à l'aide des notations classiques.

```
(*p_tot).re ← une valeur
(*p_tot).im ← une valeur
```

La notation → (flèche)

Dernier opérateur abordé pour ce qui concerne les structures, l'opérateur '→' n'est pas à proprement parler indispensable, son rôle est de procurer une syntaxe plus intuitive que le couple '*' et '.' évoqué dans le paragraphe précédent. L'utilisation de cet opérateur est des plus simples.

Exemple

Soit ptr un pointeur vers une variable de type structure, et soit ch un des champs de cette variable. Dans ce cas, deux écritures sont équivalentes :

```
(*ptr).ch    ⇔   ptr→ch
```

Ainsi, l'exemple précédent peut être écrit :

```
p_tot→im ← une valeur
p_tot→im ← une valeur
```

1.9.6 Types pointeurs et raccourcis de notation

Il est parfois pratique de disposer d'un nom de type raccourci pour désigner un pointeur vers une structure. Ce nom raccourci de type se définit avec la syntaxe suivante :

```
TYPE *type_pointeur : type
```

Exemple :

```
TYPE *ptr_comp : t_complexe
```

Le raccourci de type précédent signifie que le type `ptr_comp` désigne un pointeur vers un `t_complexe`. Ainsi, les deux définitions de variables suivantes sont équivalentes :

```
VAR *pz : t_complexe
VAR pz : ptr_comp
```

Dans les deux cas, la variable `pz` stocke l'adresse d'une valeur de type `t_complexe`.

1.9.7 Structures et fonctions

Passage de paramètres de type composé

Les structures se comportent comme les autres types en ce qui concerne les fonctions : il est possible de spécifier des entrées dont les types sont des structures, auquel cas le mécanisme de passage de paramètre reste valable. La différence réside dans le fait que plusieurs valeurs numériques doivent être transmises entre l'argument et le paramètre. Pour réaliser cette transmission, c'est en réalité l'opérateur d'affectation qui sera employé pour recopier la valeur de l'argument (expression) dans le paramètre correspondant.

Paramètres : adresses de structures

Les types composés ne diffèrent guère des types de base, mis à part l'impossibilité d'employer les opérateurs arithmétiques et logiques avec les types composés. Il en est de même avec leurs adresses : une fonction pouvant accepter une adresse en entrée (paramètres de type pointeur), elle peut donc a fortiori accepter une adresse dont le contenu est d'un type quelconque, et notamment un type composé. Il suffit dans ce cas d'utiliser la notation '$\rightarrow$' à l'intérieur de la fonction pour accéder aux champs de la structure dont l'adresse est fournie à la fonction.

Retour de valeurs de type composé

Une fonction peut tout à fait retourner une valeur de type composé, à partir du moment où ce type est défini : il peut alors être utilisé en type de sortie de fonction sans aucun problème, ce qui est d'ailleurs souvent le cas en informatique.

Retour d'adresses de structures

En synthèse des deux derniers paragraphes, une fonction peut également retourner l'adresse d'une valeur de type quelconque, notamment un type composé.

STRUCTURES SÉQUENTIELLES SIMPLES

2

RAPPELS DE COURS

2.1 LISTES LINÉAIRES

Exemple

Imaginons la gestion d'un tableau contenant les références des livres d'une bibliothèque. Ce tableau est rangé dans l'ordre alphabétique. Lorsqu'un nouveau livre est acheté, son insertion dans le tableau en respectant l'ordre requiert de déplacer toutes les références qui suivent la position d'insertion, pour dégager de la place. Le coût d'une telle opération est élevé et peut devenir prohibitif dans certains cas. Pour éviter ce type de problème, il faudrait que le passage d'une case d'un tableau à la suivante ne se fasse plus à partir d'un indice absolu qu'on incrémente, mais en notant localement dans une case du tableau l'indice de la case suivante.

On va présenter ici les **listes linéaires** qui sont la forme la plus commune d'organisation des données. On organise en liste linéaire des données qui doivent être *traitées séquentiellement*. De plus une liste est évolutive, c'est-à-dire qu'on veut pouvoir *ajouter* et *supprimer* des données.

2.1.1 Définition

Une liste linéaire est une structure de données correspondant à une suite d'éléments. Les éléments ne sont pas indexés dans la liste, mais pour chaque élément (sauf le dernier) on sait où se trouve l'élément suivant. Par conséquent, on ne peut accéder à un élément qu'en passant par le premier élément de la liste et en parcourant tous les éléments jusqu'à ce qu'on atteigne l'élément recherché.

2.1.2 Représentation

Représentation tabulaire

On peut représenter une liste par deux tableaux :

- le tableau 2.1 contient les éléments de la liste, dans un ordre quelconque.
- le tableau 2.2 est organisé de façon suivante : si la case d'indice i du premier tableau contient l'élément dont le suivant se trouve dans la case d'indice j, alors la case d'indice i de second tableau contient l'entier j.

On peut extraire les éléments du premier tableau dans l'ordre alphabétique si on connaît le point de départ : 1 dans notre cas. Ce point de départ doit évidemment être précisé à l'entrée de

Tableau 2.1

	A	d	b		\0
0	1	2	3	4	5

Tableau 2.2

	3	5	2		
0	1	2	3	4	5

chaque algorithme utilisant la représentation en question. Dans la case 1 du deuxième tableau on trouve 3, qui est l'indice du deuxième élément du premier tableau, etc. Finalement, dans la case 2 du deuxième tableau on trouve 5, et dans la case 5 du premier tableau on a le marqueur de fin.

Si on insère 'c' dans le premier tableau, on peut l'insérer dans la première case libre (c'est la case 0), et il n'y a que deux changements à faire au niveau du deuxième tableau. On met 0 dans la case 3, et on met 2 dans la case 0. Les tableaux 2.3 et 2.4 présentent les résultats.

Tableau 2.3

c	a	D	b		\0
0	1	2	3	4	5

Tableau 2.4

2	3	5	0		
0	1	2	3	4	5

Cette représentation n'est pas très intéressante : les fonctions de traitement sont relativement lourdes.

Représentation chaînée

On utilise des pointeurs pour chaîner entre eux les éléments successifs, et la liste est alors déterminée par l'adresse de son premier élément. On va définir des **enregistrements** (structures) dont un des champs est de type pointeur vers une structure chaînée du même type.

Exemple de définition d'une structure chaînée

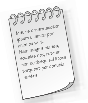
Pseudo code formel

```
STRUCTURE nœud
    nom : caractère[20]
    prenom : caractère[20]
    *suiv : nœud
TYPE *ptr_nœud : nœud
```

Implantation C

```
typedef struct nœud
{
  char nom[20];
```

```
  char prenom[20];
  struct nœud *suiv;
} nœud;
typedef nœud* ptr_nœud;
```



La liste vide est représentée par le ***pointeur NULL***.

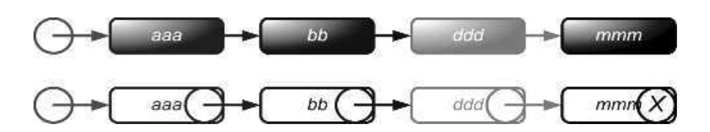


Figure 2.1 Liste (simplement) chaînée

Cette représentation n'impose pas une longueur maximum sur les listes ; elle permet de traiter facilement la plupart des opérations sur les listes : le parcours séquentiel, l'insertion et la suppression d'un élément à une place quelconque, la concaténation de deux listes, se font par une simple manipulation des pointeurs. Cependant le calcul de la longueur nécessite le parcours de toute la liste ; de même, l'accès au $k^{\text{ième}}$ élément n'est plus direct : on doit parcourir $k–1$ pointeurs à partir de la tête pour trouver le $k^{\text{ième}}$ élément.

Avant de passer aux exemples de fonctions qui traitent les listes chaînées il faut faire un petit rappel sur les variables dynamiques.

2.1.3 Variables dynamiques

- C'est une variable dont la place mémoire est allouée en cours d'exécution.
- On ne prend de la place mémoire que lorsqu'on en a besoin.
- Cette place mémoire est allouée explicitement, c'est-à-dire par une instruction du langage.
- La désallocation est également effectuée par l'intermédiaire d'une instruction.
- Il n'est donc pas nécessaire de réserver dès la compilation tout un espace mémoire.
- On utilise la place juste lorsqu'on en a besoin, puis on peut la réutiliser par autre chose.

Exemples



Les exemples sont toujours donnés en langage algorithmique.

Les données, les données modifiées et les sorties sont systématiquement précisées en préambule des déclarations.

À titre d'illustration, une réalisation (ou implantation, ou implémentation) est donnée en C99 pour chacun des types abstraits et algorithmes proposés.

Définition d'une structure de liste simplement chaînée

Pseudo code formel

```
STRUCTURE place
  info : T
  *suiv : place
TYPE *list : place
```

Implantation C

```
typedef struct place
{
  <type> content;
  struct place *succ;
} place;
typedef place *list;
```

Tester la présence d'un élément dans une liste chaînée

Spécification de l'algorithme « recherche (a) »

```
FONCTION recherche(l: liste<élément>, x: élément): booléen
VAR trouve: booléen ; p: place
DEBUT
  trouve ← faux
  SI ¬ estvide(l) ALORS
    p ← tête(l)
    TANTQUE ¬ dernier(p) ∧ trouve = faux FAIRE
      SI contenu(p) = x ALORS
        trouve ← vrai
      SINON
        p ← succ(p)  // itération vers la cellule suivante
      FINSI
    FAIT
    SI contenu(p) = x ET trouve = faux ALORS
      trouve ← vrai
    FINSI
  FINSI
  RETOURNER trouve
FIN
```

Réalisation en C

```
Données : list l : la liste de recherche, int k : l'élément recherché
Résultat : type booléen (int en C)
int containsElement(list l, int k)
{
  int found = 0;
  if (l == NULL) return found;
  while ((l->succ != NULL) & (! found))
  {
```

```
    if (l->content == k) found = 1;
    else l = l->succ;
  }
  if (l->content == k) found = 1;
  return found;
}
```

Remarque

On ne peut pas remplacer la condition `! found` dans la boucle `while` par `l->succ->content` $\neq$ `k` parce que si `l->succ` pointe sur `NULL`, `l->succ->content` n'existe pas.

Une alternative sans variable booléenne

Spécification de l'algorithme « recherche (b) »

```
FONCTION recherche(l: liste<élément>, x: élément): booléen
VAR p : place
DEBUT
  SI ¬ estvide(l) ALORS
    p ← tête(l)
    TANTQUE ¬ dernier(p) FAIRE
      SI contenu(p) = x ALORS
        RETOURNER vrai
      SINON
        p ← succ(p)  // itération vers la cellule suivante
      FINSI
    FAIT
    SI contenu(p) = x ALORS
      RETOURNER vrai
    FINSI
  FINSI
  RETOURNER faux
FIN
```

Réalisation en C (et par extension, fonction de décompte d'occurrences)

```
Données : list l : la liste de recherche, int k : l'élément recherché
Résultat : type booléen (int en C)
/**
 * Notez l'utilisation de petites fonctions utilitaires :
 * isEmptyList(l) ≡ l == NULL
 * getHeadContent(l) ≡ l->content
 * hasMoreElements(l) ≡ ! isEmptyList(nextElement(l))
 * nextElement(l) ≡ l->succ
 */
int countElementOccurrences(list l, int k)
{
  int count = 0;
  if (isEmptyList(l)) return count;
```

```
  if (getHeadContent(l) == k) count++;
  while (hasMoreElements(l))
  {
    l = nextElement(l);
    if (getHeadContent(l) == k) count++;
  }
  return count;
}
```

Créer une liste chaînée par ajouts successifs d’éléments saisis jusqu’à « fin »

Spécification de l’algorithme « création de liste en ligne de commande »

```
FONCTION créer_par_saisie(): liste
VAR l: liste ; e: élément
DEBUT
  l ← listevide()
  SAISIR(e)
  SI e ≠ fin ALORS
    cons(e, l)
  FINSI
FIN
```

Alternative

```
DEBUT
  l ← listevide()
  FAIRE
    SAISIR e
    SI e ≠ fin ALORS
      cons(e, l)
    FINSI
  TANTQUE e ≠ fin
FIN
```

Réalisation en C

Donnée modifiée : list *pl

```
/**
 * Construction d’une liste par saisie depuis l’entrée standard
 * La saisie s’achève quand l’utilisateur écrit ’fin’
 * La saisie d’un terme qui n’est pas un entier ni fin
 * est interprétée comme la saisie de ’0’
 */
int newListFromStandardInput(list *pl)
{
  *pl = NULL;
  int size = 1;
  char input[10];
  printf("Saisissez des entiers puis tapez ’fin’ :\n");
  printf("%d> ", size++);
```

```
  scanf("%s", input);
  if (strcmp("fin", input) == 0) return 0;
// allocation mémoire et rattachement de la tête
  (*pl) = malloc(sizeof(listNode));
  list l = *pl;
  l->contenu = atoi(input);    // affectation du contenu
  l->prev = NULL;              // en + pour liste doublement chaînée
  printf("%d> ", size++);
  scanf("%s", input);
  list prev_l;
  while (strcmp("fin", input) != 0) //(contenu != 0)
  {
    prev_l = l;
    l = l->succ;                 // itération
    l = malloc(sizeof(listNode));  // allocation
    l->contenu = atoi(input);      // affectation du contenu
    // important, car la valeur par défaut n'est pas forcément NULL !!!
    prev_l->succ = l;
    l->prev = prev_l;      // en + pour liste doublement chaînée
    printf("%d> ", size++);
    scanf("%s", input);
  }
  l->succ = NULL;
  return (size - 2);
}
```

Supprimer un élément de la liste

Spécification de l'algorithme « supprimer un élément d'une liste »

```
FONCTION supprimer(l: liste<élément>, x: élément): booléen
VAR p, prec, cour: place
VAR ok: booléen
DEBUT
  ok ← faux
  p ← tête(l)
  SI contenu(p) = x ALORS
    prec ← p
    p ← succ(p)
    ok ← vrai
    { suppression du premier élément }
    LIBERER(prec)
  SINON
    prec ← p
    cour ← succ(p)
    TANTQUE ok = faux ∧ cour ≠ listevide() FAIRE
      { cour pointe sur celui à supprimer, }
      { prec pointe sur le précédent }
      SI contenu(cour) = x ALORS
        ok ← vrai
```

```
        { création du lien pour supprimer un élément }
        { situé entre les deux qui sont liés }
        succ(prec) ← succ(cour)
        LIBERER(cour)
      SINON
        { mémorisation de la place d'élément précédent }
        { l'élément à tester }
        prec ← cour
        cour ← succ(cour)
      FINSI
    FAIT
  FINSI
  RETOURNER ok
FIN
```

Réalisation en C (avec extensions pour les listes circulaires et doublement chaînées)

```
Donnée : int x : l'élément à supprimer
Donnée modifiée : list* pl : la liste d'où supprimer l'élément
Résultat : type booléen (int en C)
int removeFirst(list* pl, int x)
{
  list prec, cour;
  int ok = 0;
  if (*pl == NULL) return ok;
  int circular = isCircular(*pl);  // À vous de jouer
  int perimeter = getCLength(*pl); // Idem
  if ((*pl)->content == x) // si le x est le premier élément..
  {
    prec = *pl;        // la tête est notée 'prec'
    *pl = (*pl)->succ; // on déplace le début de liste au suivant
    if (*pl != NULL) (*pl)->prev = NULL; // pour la doublement chaînée
    // en + pour la circulaire qui pendant un instant ne l'est plus :
    if (circular)
    {
      if ((*pl)->succ == prec) (*pl)->succ = *pl;
      else
      {
        list last = *pl;
        while (last->succ != prec) last = last->succ;
        last->succ = *pl;
      }
    }
    ok = 1;
    free(prec);     // on libère la mémoire de l'ex tête
  }
  else
  {
    prec = *pl;     // la tête est notée prec
```

```
      cour = (*pl)->succ;
      while (((! ok) && (cour != NULL)) &
             (! circular || (circular & (--perimeter > 0))))
      {
        if (cour->content == x) // removeFirst et terminaison
        {
          ok = 1;
          prec->succ = cour->succ;
          // en + pour la liste doublement chaînée :
          if (cour->succ != NULL) cour->succ->prev = prec;
          free(cour);
        }
        else          // itération
        {
          prec = cour;
          cour = cour->succ;
        }
      }
    }
    return ok;
  }
```

2.1.4 Variantes d'implantation des listes

Il existe d'ailleurs de nombreuses variantes de la représentation des listes à l'aide de pointeurs.

Cellule racine

Il est parfois commode de définir une liste non pas comme un pointeur sur une cellule, mais par un bloc de cellules du même type que les autres cellules de la liste. Ce bloc ne contient que l'adresse du premier élément de la liste. L'utilisation d'un tel bloc permet d'éviter un traitement spécial pour l'insertion et la suppression en début de liste.

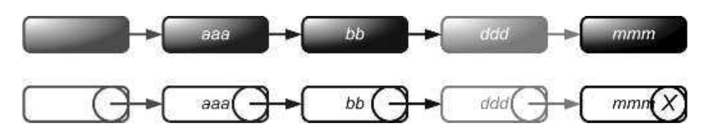


Figure 2.2 **Liste (linéaire simplement) chaînée avec cellule racine**

Liste circulaire

On peut aussi créer des **listes circulaires** : on remplace, dans la dernière place de la liste, le pointeur à NULL par un pointeur vers la tête de la liste ; de plus si l'on choisit la dernière place comme point d'entrée dans la liste, on retrouve la tête de liste en parcourant un seul lien.

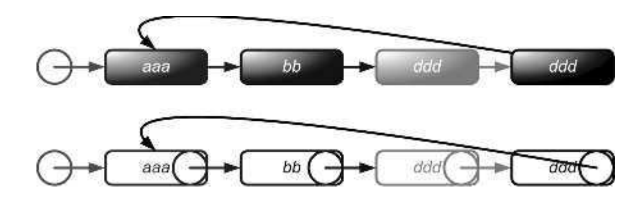


Figure 2.3 Liste (simplement) chaînée circulaire

Liste doublement chaînée (ou chaînée bidirectionnelle)

Dans la représentation classique, le parcours des listes est orienté dans un seul sens : du premier élément vers le dernier élément. Cependant de nombreuses applications nécessitent de parcourir les listes à la fois vers l'avant et vers l'arrière, et dans ce cas on peut faciliter le traitement en rajoutant des « pointeurs arrière », ce qui augmente évidemment la place mémoire utilisée. On obtient alors une **liste doublement chaînée** : chaque place comporte un pointeur vers la place suivante et un pointeur vers la place précédente.

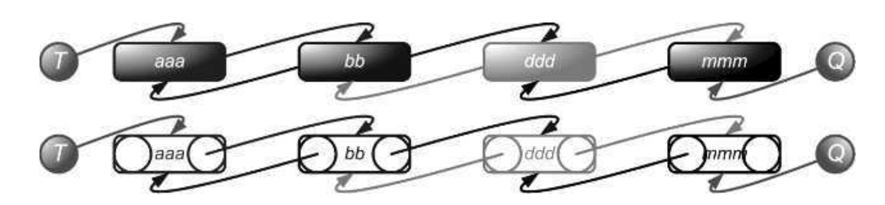


Figure 2.4 Liste (linéaire) doublement chaînée

Définition d'une structure de liste doublement chaînée

Pseudo code formel

```
STRUCTURE node
  content : T
  *succ, *prev : node
TYPE *list : node
```

Implantation C

```
typedef struct node
{
  <type> content;
  struct node *succ, *prev;
} node;
typedef node *list;
```

Afficher le contenu d'une liste à l'envers

Spécification abstraite

```
FONCTION afficher(l: liste)
DEBUT
  l ← tête(l)
  TANTQUE ¬ dernier(l) FAIRE
    l ← succ(l)
  FAIT
  TANTQUE ¬ premier(l) FAIRE
    AFFICHER contenu(l)
    l ← prev(l)
  FAIT
FIN
```

Réalisation en C

Donnée : list l : la liste à afficher

```
void reversePrintList(list l)
{
  int k = 1;
  while (l->succ != NULL)
  {
    l = l->succ;
    k++;
  }
  while (l != NULL)
  {
    printf("\tList[%d] = %d\n", k--, l->content);
    l = l->prev;
  }
}
```

ÉNONCÉS DES EXERCICES ET DES PROBLÈMES

EXERCICES

Exercice 2.1 Rechercher l'élément maximal d'une liste **

Écrire deux algorithmes, l'un itératif, l'autre récursif, qui permettent de déterminer la valeur maximale d'une liste chaînée d'entiers positifs.

Exercice 2.2 Concaténer deux listes *

On considère deux listes chaînées L_1 et L_2 dont les éléments sont des entiers. Écrire un algorithme qui rattache la liste L_2 à la suite de la liste L_1. Tous les cas particuliers doivent être pris en compte.

Exercice 2.3 Extraire deux listes à partir d'une liste ***

Soit une liste de nombres relatifs. Écrire un algorithme permettant de séparer cette liste en deux listes : la première ne comportant que des entiers positifs ou nuls, et la seconde ne comportant que des nombres négatifs.

Exercice 2.4 Permuter deux places d'une liste ***

Écrire un algorithme sur une liste simplement chaînée qui échange les positions des nœuds données par deux pointeurs `t` et `v`.

Exercice 2.5 Supprimer des éléments **

Soit L une liste chaînée. Écrire trois algorithmes tels que :

- dans le premier on supprime toutes les occurrences d'un élément donné `x` ;
- dans le deuxième, on ne laisse que les *k* premières occurrences de cet élément et on supprime les suivantes ;
- dans le troisième, pour chaque élément de la liste, on ne laisse que sa première occurrence.

Concevez le second algorithme en adaptant le premier, puis concevez le dernier en exploitant le second.

Exercice 2.6 Inverser une liste **

Écrire un algorithme qui inverse une liste chaînée sans recopier ses éléments. Réaliser une version itérative et une autre récursive.

Exercice 2.7 Construire une liste à partir d'une source de données *

Concevoir un premier algorithme qui construit une liste chaînée linéaire à partir d'un tableau, et un second qui duplique une liste chaînée existante.

Exercice 2.8 Refermer une liste sur elle-même *

Concevoir un algorithme qui transforme une liste simplement chaînée linéaire en liste simplement chaînée circulaire.

Exercice 2.9 Effectuer et retourner deux calculs sur une liste **

Concevoir un algorithme qui calcule et retourne respectivement le produit des éléments positifs et celui des éléments négatifs d'une liste simplement chaînée d'entiers.

Exercice 2.10 Couper une liste en deux **

Concevoir un algorithme qui sépare une liste simplement chaînée d'entiers en deux listes :

- A – à partir d'un pointeur sur le maillon qui devient tête de la seconde liste de sortie ;
- B – juste devant le premier maillon contenant une valeur donnée *x* ;
- C – à partir de la $k^{ième}$ position (c'est-à-dire *k* maillons dans la première liste de sortie et le reste dans la seconde) ;
- D – juste devant le maillon contenant la valeur minimale de la liste ;
- E – telles que les deux listes de sortie aient même longueur *n*, ou que la première soit de longueur *n+1* et la seconde de longueur *n*.

Exercice 2.11 Supprimer un sous-ensemble d'une liste *

Concevoir un algorithme qui supprime :

- A – un maillon sur deux, en commençant par la tête de liste,
- B – les maillons contenant des éléments impairs,
- C – les maillons contenant des éléments pairs,
- D – les maillons contenant des éléments supérieurs à un seuil donné,
- E – les maillons contenant des éléments inférieurs à un seuil donné,

d'une liste simplement chaînée (d'entiers).

PROBLÈME

Problème 2.1 Saisir, enregistrer puis évaluer un polynôme***

Enregistrer les coefficients et les exposants d'un polynôme dans une liste chaînée. Évaluer ce polynôme pour une valeur donnée de *x*.

Utiliser au moins deux modules dans ce programme :

- l'un pour construire la liste sur la base des coefficients et des exposants, qui sont saisis au clavier ;
- et le second afin d'évaluer le polynôme pour la valeur *x*, également saisie au clavier.

Corrigés des exercices et des problèmes

EN PRÉAMBULE

Pour la réalisation en C de tous les algorithmes spécifiés ci-dessous, on définit la structure de liste chaînée suivante dont on précisera au cas pas cas, le type `<element>`. Les structures utilisées pour les spécifications des algorithmes portent le même nom et sont construites de la même manière.

```
typedef struct listNode
{
  <element> content;
  struct place *succ;  // listes simplement chaînées
  struct place *prev;  // listes doublement chaînées
} listNode;
typedef listNode* list;
typedef list* plist;
#define ERR -1  // code d'erreur (lisibilité des exemples)
```

Les quelques implantations C de méthodes élémentaires de manipulation de la liste chaînée conformes au contrat du type abstrait, pourront s'avérer utiles dans l'écriture de vos programmes d'implantation C de vos algorithmes, notamment en épurant votre code d'une trop grande quantité

de '->', '*' et '&' qui nuisent à leur lisibilité. Vous pourrez ainsi vous rapprocher de la vision algorithmique du « qu'est-ce que ça fait ? » plutôt que de la vision programmatique du « comment ça se fait ? ».

```
<element> content(list l) {return (l->content);}

int isempty(list l) {return (l == NULL);}

list succ(list l) {return l->succ;}

int islast(list l) {return isempty(succ(l));}
```

pour les solutions algorithmiques, les fonctions sont les suivantes :

```
FONCTION content(l : list)
DEBUT
  RETOURNER l→content
FIN
FONCTION isempty(l : list)
VAR vide : booléen
DEBUT
  SI(l=NULL)ALORS
    vide ← vrai
  SINON
    vide ← faux
  FINSI
  RETOURNER vide
FIN
FONCTION succ(l : list) : list
DEBUT
  RETOURNER l→succ
FIN
FONCTION islast(l : list)
DEBUT
  RETOURNER isempty(succ(l))
FIN
```

Enfin, le constructeur suivant pourra vous aider à simplifier vos écritures pour l'allocation d'un nouveau nœud de liste :

```
list newSingletonList(int content)
{
  list l = malloc(sizeof(listNode));  // allocation mémoire
  l->content = content;               // affectation du contenu
  l->succ = NULL;
  return l;
}
```

Et son équivalent en pseudo langage :

```
FONCTION newSingletonList(content : entier) : list
VAR nouveau : list
DEBUT
  RESERVER(nouveau)
  nouveau→content ← content
  nouveau→succ ← NULL
  RETOURNER nouveau
FIN
```

Corrigés des exercices

Exercice 2.1 Rechercher l'élément maximal d'une liste

Spécification de l'algorithme itératif

C'est une simple adaptation de l'algorithme « recherche (b) » du rappel de cours vu plus haut.

Entrée : liste de positifs (non vide)

Sortie : un positif

```
FONCTION maxOf(l: liste<positif> ≠ ∅): positif
VAR max: positif, p: place
DEBUT
  p ← tête(l)
  max ← contenu(p)
  TANTQUE ¬ dernier(p) FAIRE
    p ← succ(p)
    SI contenu(p) > max ALORS
      max ← contenu(p)
    FINSI
  FAIT
  RETOURNER max
FIN
```

Réalisation en C de l'algorithme itératif

Ici, le champ content est de type `unsigned`.

```
int maxOf(list<unsigned> l)
{
  if (l == NULL) return ERR;
  int max = l->content;    // valeur en première place
  while (l->succ != NULL)
  {
    l = l->succ;  // itération de la liste
    if (l->content > max) max = l->contenu;
  }
  return max;
}
```

Et réécrite avec les méthodes utilitaires données en introduction :

```
int maxOf(list<unsigned> l)
{
  if (isempty(l)) return ERR;
  int max = content(l);  // valeur en première place
  while (! islast(l))
  {
    l = succ(l);  // itération de la liste
    if (content(l) > max) max = content(l);
  }
  return max;
}
```

Spécification de l'algorithme récursif

Entrée : liste de positifs (non vide)
Sortie : un positif

```
FONCTION maxOf(l: liste<positif> ≠ ∅): positif
VAR max, contenu: positif
DEBUT
  RETOURNER maxOfRec(tête(l))
FIN
FONCTION maxOfRec(p: place): positif
VAR max, contenu: positif
DEBUT
  contenu ← contenu(p)
  SI dernier(p) ALORS
    RETOURNER contenu
```

```
    FINSI
    max ← maxOfRec(succ(p))
    SI contenu > max ALORS
      RETOURNER contenu
    FINSI
    RETOURNER max
  FIN
```

Réalisation en C de l'algorithme récursif

```
int maxOfRec(list<unsigned> l)
{
  if (l == NULL) return ERR;
  int content = l->content;
  if (l->succ == NULL) return content;
  int max = maxOfRec(l->succ);
  if (content > max) return content;
  return max;
}
```

Et réécrite avec les méthodes utilitaires données en introduction :

```
int maxOfRec(list<unsigned> l)
{
  if (isempty(l)) return ERR;
  int content = content(l);
  if (islast(l)) return content;
  int max = maxOfRec(succ(l));
  if (content > max) return content;
  return max;
}
```

Exercice 2.2 Concaténer deux listes

Étude

La contrainte d'entiers positifs du premier exercice disparaît, mais pas le cas des listes vides. Nous avons quatre cas à considérer :

- l_1 et l_2 vides,
- l_1 vide mais pas l_2,
- l_2 vide mais pas l_1,
- ni l_1, ni l_2 vides.

Si l_2 est vide, on laisse l_1 inchangée. Si l_1 est vide mais pas l_2, on affecte directement l_2 à l_1. Si ni l_1 ni l_2 ne sont vides, on parcourt l_1 jusqu'à sa dernière cellule et on y rattache l_2.

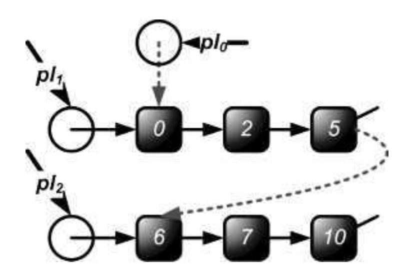


Figure 2.5 Concaténation standard

Enfin, si on va un peu trop vite dans l'étude, on peut oublier le cas $l_1 = l_2$ (c'est-à-dire même liste et non pas, listes dont les valeurs sont égales), cas particulier qui croise le premier et le dernier des quatre cas. Ce cas donne une bonne illustration de la différence entre copies et références.

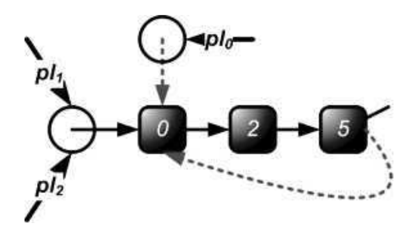


Figure 2.6 Concaténation circulaire

Ce dernier cas d'utilisation de la fonction de concaténation revient, si on l'autorise, à transformer la liste chaînée linéaire en liste chaînée circulaire.

Autorisons-le, en notant que nous avons alors un constructeur de liste circulaire par fermeture d'une liste chaînée linéaire (nous utiliserons cette facilité dans le problème 3.1 (problème de Joseph)).

Pour la concaténation, une approche itérative s'impose d'elle-même, un procédé récursif n'ayant pas grand intérêt pour une simple jonction.

Nous travaillons par référence, c'est-à-dire qu'il n'y a pas de copie des deux listes (on « rattache » les deux listes).

Spécification de l'algorithme

Entrées modifiées : deux listes d'entiers

```
FONCTION concat(l₁, l₂: liste<entier>)
VAR p₁, p₂ : place
DEBUT
  SI ¬ estvide(l₁) ALORS
    SI estvide(l₂) ALORS
      l₁ ← l₂
    SINON
```

```
      p1 ← tête(l1)
      TANTQUE ¬ dernier(p1) FAIRE
        p1 ← succ(p1)
      FAIT
      succ(p1) ← tête(l2)
    FINSI
  FINSI
FIN
```

Réalisation en C

```
void concat(list *pl1, list *pl2)
{
  if (*pl2 == NULL) return;  // ras, l1 soit nulle ou non
  if (*pl1 == NULL)          // affectation directe de l2 à l1
  {
    *pl1 = *pl2;
    return;
  }
  // sinon :
  list l = *pl1;
  while (l->succ != NULL) l = l->succ;  // itération l1
  l->succ = *pl2;                       // rattachement de l2
}
```

Et réécrite avec les méthodes utilitaires données en introduction :

```
void concat(list *pl1, list *pl2)
{
  if (isempty(*pl2)) return; // ras, l1 soit nulle ou non
  if (isempty(*pl1))         // affectation directe de l2 à l1
  {
    *pl1 = *pl2;
    return;
  }
  // sinon :
  list l = *pl1;
  while ((! islast(l)) l = succ(l);  // itération l1
  l->succ = *pl2;                    // rattachement de l2
}
```

Exercice 2.3 Extraire deux listes à partir d'une liste

Étude

Plusieurs approches sont possibles selon que :

- On travaille à partir de la liste de départ en la préservant (copie des éléments), ou bien en recyclant ses cellules (réutilisation des éléments).
- On réalise une seule fonction complexe qui produit simultanément les deux listes, ou bien deux fonctions simples spécialisées appelées successivement.
- On élabore une structure complexe nouvelle pour combiner les deux listes produites dans l'unique sortie de la fonction, ou bien on passe deux pointeurs respectivement sur les deux listes (pointeurs sur pointeurs) initialement non initialisés (paramètres inout).

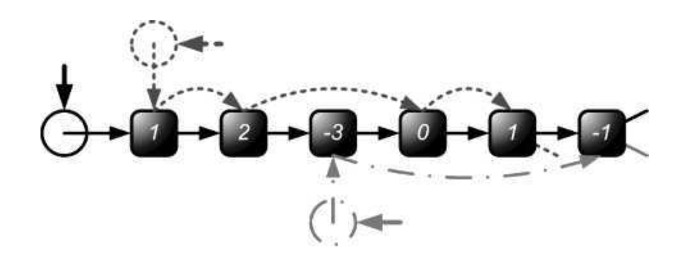


Figure 2.7 Déconstruction d'une liste avec reconstruction sous forme de deux listes

Nous nous proposons d'utiliser le même algorithme itératif pour réaliser l'opération, que ce soit par copie ou par référence. Cet algorithme consiste à parcourir la liste de départ et à aiguiller chaque nouvel élément rencontré vers l'une ou l'autre des deux listes selon sa valeur, en les complétant ainsi progressivement jusqu'à épuisement de la liste de départ. Il existe d'autres solutions.

Spécification de l'algorithme

Entrées modifiées :

- La liste d'entier initiale
- Les deux listes d'entiers produites

```
FONCTION separer(l0, l1, l2: liste<entier>)
VAR p0, p1, p2: place
DEBUT
  SI ¬ estvide(l0) ALORS
    p0 ← tête(l0)
    SI contenu(p0) ⩾ 0 ALORS
      p1 ← tête(l1) ← tête(l0)
    SINON
    p2 ← tête(l2) ← tête(l0)
    FINSI
    TANTQUE ¬ dernier(p0) FAIRE
```

```
      p0 ← succ(p0)
      SI contenu(p0) ⩾ 0 ALORS
        SI estvide(l1) ALORS
          p1 ← tête(l1) ← p0
        SINON
          p1 ← succ(p1) ← p0
        FINSI
      SINON
        SI estvide(l2) ALORS
          p2 ← tête(l2) ← p0
        SINON
          p2 ← succ(p2) ← p0
        FINSI
      FINSI
    FAIT
  FINSI
FIN
```

Réalisation en C

```
void separate(list *pl0, list *pl1, list *pl2)
{
  liste l0 = *pl0, l1 = NULL, l2 = NULL;
  if (l0 == NULL) return;     // si l0 vide ne rien faire
  if (l0->content >= 0) l1 = *pl1 = *pl0; // tête de l1
  else l2 = *pl2 = *pl0;                  // tête de l2
  while (l0->succ != NULL)  // itération de l0
  {
    l0 = l0->succ;          // itération de l0
    if (l0->content >= 0)   // transfert vers l1
    {
      if (l1 == NULL) *pl1 = l1 = l0;  // tête de l1
      else                  // au-delà de la tête
      {
        l1->succ = l0;      // complète l1
        l1 = l1->succ;      // itération de l1
      }
    }
    else                    // transfert vers l2
    {
      if (l2 == NULL) *pl2 = l2 = l0;  // tête de l2
      else                  // au-delà de la tête
      {
        l2->succ = l0;      // complète l2
```

```
        l2 = l2->succ;      // itération de l1
      }
    }
  }
  // important pour la clôture des listes l1 et l2 :
  l1->succ = NULL;          // finalisation de l1
  l2->succ = NULL;          // finalisation de l2
  *pl0 = NULL;              // libération de l0 (opt.)
}
```

Exercice 2.4 Permuter deux places d'une liste

Étude

Il convient de prévoir un code d'erreur pour les cas suivants :

- Liste vide mais *t* ou *v* non null.
- Liste non vide, mais *t* ou *v* pointant nul ou pointant un élément n'appartenant pas à la liste.
- La fonction retournera –1 en cas d'erreur, 0 sinon.

Pour le reste, l'idée est d'identifier les deux éléments, et d'interchanger (en utilisant une variable temporaire) les pointeurs de leurs prédécesseurs respectifs et de leurs successeurs respectifs.

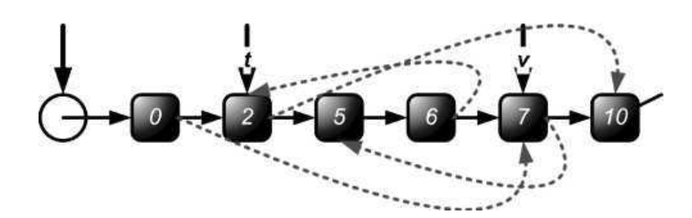


Figure 2.8 Permutation : le cas général...

Les cas suivants feront l'objet d'un traitement spécial :

- Les pointeurs *t* et *v* sont égaux : c'est l'opération identité, il n'y a rien à faire sinon ne rien faire.
- L'un de deux pointeurs ou les deux sont respectivement tête ou queue (début ou fin de liste).

Les deux pointeurs identifient des places contigües (cf. figure 2.9 ce cas est « dangereux » si traité comme précédemment).

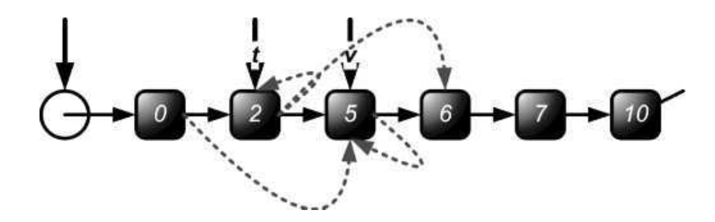


Figure 2.9 ... avec son exception dangereuse en l'absence de traitement spécifique

Spécification de l'algorithme

```
FONCTION swap(modif pl : list, modif t, v : place) : entier
VAR resultat : entier ; term_l, term_t, term_v, l : list
VAR prec_t, succ_t, prec_v, succ_v : list
DEBUT
  SI (pl = NULL) ALORS
    SI (t = NULL) ET (v = NULL) ALORS
      RETOURNER 0
    SINON
      RETOURNER -1
    FINSI
  SINON
    SI (t = NULL) OU (v = NULL) ALORS
      RETOURNER -1
    FINSI
      term_l ← dernierElt(*pl)
      term_t ← dernierElt(t)
      term_v ← dernierElt(v)
    SI (¬ ((term_t = term_l) ET (term_v = term_l))) ALORS
      RETOURNER -1
    FINSI
  FINSI
  SI (t = v) ALORS
    RETOURNER 0
  FINSI
  l = pl;
  trouverPrecEtSucc(l, t, &prec_t, &succ_t)
  trouverPrecEtSucc(l, v, &prec_v, &succ_v)
  SI ((prec_v = t) OU (prec_t = v)) ALORS
    SI prec_v = t ALORS
      SI prec_t = NULL ALORS
        *pl ← v
      SINON
        prec_t→succ ← v
      FINSI
      v→succ ← t
      t→succ ← succ_v
    SINON
      SI prec_v = NULL ALORS
        *pl ← t
      SINON
        prec_v→succ ← t
      FINSI
```

```
      t→succ ← v
      v→succ ← succ_t
    FINSI
  SINON
    SI ((prec_t = NULL) OU (prec_v = NULL) ALORS
      SI prec_t = NULL ALORS
        *pl ← v
        prec_v→succ ← t
      FINSI
      SI prec_v = NULL ALORS
        *pl ← t
        prec_t→succ ← v
      FINSI
    SINON
      prec_v ← t
      prec_t ← v
    FINSI
    t→succ ← succ_v
    v→succ ← succ_t
  FINSI
FIN
```

Réalisation en C de la fonction principale

```
// Permutation de deux éléments
int swap(list *pl, place *t, place *v)
{
  if (*pl == NULL)
  {
    if ((t == NULL) && (v == NULL)) return 0; // tout va bien !!
    else return -1;
  }
  else
  {
    if ((t == NULL) || (v == NULL)) return -1; // non consistant !!
    // Maintenant test d'appartenance
    // on utilise un truc : même terminal !
    list term_l = lastElement(*pl);
    list term_t = lastElement(t);
    list term_v = lastElement(v);
    if (!((term_t == term_l) && (term_v == term_l))) return -1;
  }
  // préconditions vérifiées : on peut commencer
  if (t == v) return 0; // identité -- tout va bien !!
```

```
// note : le cas du singleton est implicitement déjà traité :
// on travaille maintenant sur une liste l multiple
// sinon, cas général :
list l = *pl;
list prec_t, succ_t;
getPrecAndSucc(l, t, &prec_t, &succ_t);
l = *pl;
list prec_v, succ_v;
getPrecAndSucc(l, v, &prec_v, &succ_v);
// cas de contigüité ->[v]->[t]-> ou ->[t]->[v]->
if ((prec_v == t) || (prec_t == v))
{
  // cas : ->[t]->[v]-> => ->[v]->[t]->
  if (prec_v == t)
  {
    // cas .->[t] => .->[v] vs autres cas
    if (prec_t == NULL)
    {
      printf("case : .->[t]->[v]-> => .->[v]->[t]->\n");
      *pl = v;
    }
    else
    {
      printf("case : [..]->[t]->[v]-> => [..]->[v]->[t]->\n");
      prec_t->succ = v;
    }
    v->succ = t;
    t->succ = succ_v;
  }
  // cas : ->[v]->[t]-> => ->[t]->[v]->
  else
  {
    // cas .->[v] => .->[t] vs autres cas
    if (prec_v == NULL)
    {
      printf("case : .->[v]->[t]-> => .->[t]->[v]->\n");
      *pl = t;
    }
    else
    {
      printf("case : [..]->[v]->[t]-> => [..]->[t]->[v]->\n");
      prec_v->succ = t;
    }
```

```
      t->succ = v;
      v->succ = succ_t;
    }
  }
  // cas ->[v]->..->[t]-> ou ->[t]->..->[v]->
  else
  {
    // cas .->[t] => .->[v] vs autres cas
    if ((prec_t == NULL) || (prec_v == NULL))
    {
      if (prec_t == NULL)
      {
        printf("case : .->[t]->[..]->[v]-> => .->[v]->[..]->[t]->\n");
        *pl = v;
        prec_v->succ = t;
      }
      if (prec_v == NULL)
      {
        printf("case : .->[v]->[..]->[t]-> => .->[t]->[..]->[v]->\n");
        *pl = t;
        prec_t->succ = v;
      }
    }
    else
    {
      printf("case : [..]->[t/v]->[..]->[v/t]-> =>
[..]->[v/t]->[..]->[t/v]->\n");
      prec_v = t;
      prec_t = v;
    }
    // et dans tous les cas :
    t->succ = succ_v;
    v->succ = succ_t;
  }
  return 0; // that's all folks !!
}
```

Fonction déléguée de récupération du prédécesseur et du successeur d'un nœud

```
// (fonction sécurisée par l'appelant :
// paramètres supposés cohérents entre eux)
int getPrecAndSucc(list l, list t, list* prec, list* succ)
{
  if (l == t)       // cas de t en tête
```

```
    {
      *prec = NULL;
      *succ = l->succ;
      return;
    }
    else          // cas de t en milieu ou en queue
    {
      while (l->succ != NULL)
      {
        if (l->succ == t)
        {
          *prec = l;
          *succ = l->succ->succ;
          return;
        }
        else l = l->succ;
      }
    }
  }
```

Exercice 2.5 Supprimer des éléments

Étude

Nous proposons de réaliser le second algorithme en premier lieu, puis de construire le premier par un appel récursif du second avec pour condition d'arrêt, un dernier appel qui laisse la liste inchangée.

Pour supprimer un élément, il s'agit de repérer l'élément, et de court-circuiter celui-ci en reliant le pointeur `succ` de son prédécesseur directement sur son successeur.

Algorithme récursif

- *Retrait de toutes les occurrences d'un élément*

Entrée : x, l'entier dont on supprime toutes les occurrences dans la liste.

Entrée modifié : la liste dont on supprime des occurrences de x.

Sortie : un code de statut d'exécution

- `SUPPRESSION (1)` : il y a eu au moins un retrait effectif.
- `IDENTITE (0)` : il n'y a eu aucun retrait car la liste ne comporte aucun x.

Cette fonction utilise un type énuméré (énumération) qui est un type défini par les valeurs que peuvent prendre les variables de ce type. Ainsi, dans la fonction suivante, les variables *`statut`* et *`st2`* ne peuvent prendre que les valeurs *`SUPPRESSION`* ou *`IDENTITE`*.

```
FONCTION supprimerOccurrences(*l: liste<entier>, x: entier)
VAR statut, st2: énumération {SUPPRESSION, IDENTITE}; e: entier
DEBUT
```

```
  statut ← IDENTITE
  SI ¬ estvide(l) ALORS
    SI premier(l) = x ALORS
      l ← fin(l)
      statut ← SUPPRESSION
      SI ¬ estvide(l) ALORS
         supprimerOccurrences(l, x)
      FINSI
    SINON
      e ← premier(l)
       l ← fin(l)
      SI ¬ estvide(l) ALORS
         st2 ← supprimerOccurrences(l, x)
         SI statut = IDENTITE ALORS
           statut ← st2
         FINSI
      FINSI
       l ← cons(e, l)
     FINSI
   FINSI
  RETOURNER statut
FIN
```

- *Retrait de toutes les occurrences d'un élément après la* $k^{ième}$

Très semblable au précédent.
Entrées : x, l'entier dont on supprime toutes les occurrences sauf les k premières dans la liste, k le nombre d'occurrences qu'on laisse sauves.
Entrées modifiées : la liste dont on supprime certaines occurrences de x.
Sortie : un code de statut d'exécution

- `SUPPRESSION (1)` : il y a eu au moins un retrait effectif.
- `IDENTITE (0)` : il n'y a eu aucun retrait car la liste ne comporte aucun x.

```
FONCTION supprOccurAprèsKième(*l: liste<entier>, x: entier, k: entier)
VAR statut, st2: énumération {SUPPRESSION, IDENTITE}; e: entier;
DEBUT
  statut ← IDENTITE
  SI ¬ estvide(l) ALORS
    SI premier(l) = x ALORS
      SI k < 1 ALORS
         l ← fin(l)
        statut ← SUPPRESSION
       SINON
        e ← premier(l)
         l ← fin(l)
```

```
          k ← k -- 1
        FINSI
       SI ¬ estvide(l) ALORS
          supprOccurAprèsKième(l, x, k)
       FINSI
       SI k > 0 ALORS
          l ← cons(e, l)
        FINSI
     SINON
       e ← premier(l)
        l ← fin(l)
       SI ¬ estvide(l) ALORS
         st2 ← supprOccurAprèsKième(l, x, k)
          SI statut = IDENTITE ALORS
            statut ← st2
          FINSI
       FINSI
        l ← cons(e, l)
      FINSI
    FINSI
  RETOURNER statut
FIN
```

Exercice 2.6 Inverser une liste

Étude

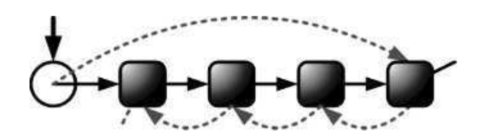


Figure 2.10 **Inversion d'une liste**

Deux approches, l'une itérative, l'autre récursive.

Algorithme itératif

```
FONCTION reverse(*pl : list)
VAR prev, curr, succ : list
DEBUT
  SI *pl=NULL ALORS   // cas de la liste vide
    RETOURNER
  FINSI
  SI (*pl) →succ = NULL ALORS // cas de la liste à un seul élément
```

```
    RETOURNER
  FINSI
  prev ← *pl
  curr ← prev→succ
  SI curr→succ = NULL ALORS
    curr→succ ← prev
    prev→succ ← NULL
    *pl ← curr
    RETOURNER
  SINON
    prev→succ ← NULL
  FINSI
  TANTQUE curr→succ ≠ NULL FAIRE
    succ ← curr→succ
    curr→succ ← prev
    prev ← curr
    curr ← succ
  FAIT
  curr→succ ← prev
  *pl ← curr
FIN
// Inversion d'une liste (utilisée pour l'exo 1.4 (normalisation))
void reverse(list *pl)
{
  if (*pl == NULL) return;           // liste vide
  if ((*pl)->succ == NULL) return;   // singleton
  list prev = *pl;                   // tête
  list curr = prev->succ;            // 2d élément
  if (curr->succ == NULL)            // si cas à 2 éléments
  {
    curr->succ = prev;               // pointage du 2d élt sur le 1er
    prev->succ = NULL;               // et du 1er sur NULL (queue)
    *pl = curr;                      // le 2d devient tête
    return;                          // terminé
  }
  else                               // sinon, au moins 3 elts :
  {
    prev->succ = NULL;               // pointage du 1er->NULL (queue)
  }
  list succ;
  while (curr->succ != NULL)         // Expl. sur 1ère itération
  {
    succ = curr->succ;               // sauv. du succ (le 3ème élt)
```

```
    curr->succ = prev;             // pointage du 2d elt sur le 1er
    prev = curr;                   // le nouveau prev c'est le 2d
    curr = succ;                   // le nouveau curr, c'est le 3ème
  }
  // si par exemple le 3ème n'a pas de successeur :
  // c'est la queue (pas d'autre passage dans la boucle) :
  curr->succ = prev;               // pointage du 3ème sur le 2d
  *pl = curr;                      // le 3ème (dernier) devient tête
}
```

Algorithme récursif

```
FONCTION esreverse(*pl : list) : list*
VAR head : list
DEBUT
  SI *pl=NULL ALORS   // cas de la liste vide
    RETOURNER
  FINSI
  SI (*pl) →succ = NULL ALORS // cas de la liste à un seul élément
    RETOURNER
  FINSI
  head ← *pl
  *esreverse(&(head→succ)) → succ ← head
  *pl ← head→succ
  head→succ ← NULL
  RETOURNER &head
FIN
// reverse2 en méthode récursive
list * esreverse(list* pl)
{
  if (*pl == NULL) return;             // cas liste vide
  if ((*pl)->succ == NULL) return pl;  // cas singleton
  // autres cas :
  list head = *pl;
  (*esreverse(&(head->succ)))->succ = head;
  *pl = head->succ;
  head->succ = NULL;
  return &head;
}
```

Exercice 2.7 Construire une liste à partir d'une source de données

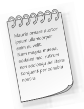
• *Création depuis un tableau d'entiers*

Spécification de l'algorithme

```
FONCTION newListFromArray(content : entier[], longueur : entier) : list
{
VAR listenouv, courant : list ; i : entier
DEBUT
  SI (longueur < 1) ALORS // la liste sera vide
    RETOURNER NULL
  FINSI
  listenouv ← newSingletonList(content[0]) // construction de la tête
  courant ← listenouv
  // ajout des autres places à la suite
  POUR i DE 1 A longueur-1 FAIRE
    courant→succ ← newSingletonList(content[i])
    courant ← courant→succ    // passage au suivant
  FAIT
  RETOURNER listenouv // retour de la tête, qui représente la liste
FIN
```

Implantation C

```
list newListFromArray(int* content, int length)
{
  if (length < 1) return emptyList(); // cas de la liste vide
  // sinon : initialisation de la chaîne avec la création de la tête
  list l = newSingletonList(content[0]);
  // déclaration et initialisation d'un variable d'itération de liste
  list pp = l;
  // itération de la 1ère place à la dernière ([longueur-1] intervalles)
  int i;
  for (i = 1; i < length; i++)
  {
    pp->succ = newSingletonList(content[i]);
    //pp->succ->prev = pp;        // chaînage arrière (opt.)
    pp = pp->succ;                // itération de la liste
  }
  return l;// retour du pointeur sur la tête, lequel représente la liste
}
```



Il est possible mais risqué, par l'astuce suivante, d'éviter le passage du paramètre qui indique la longueur du tableau source : `size_t longueur = sizeof(contenu) / sizeof(int);`

- *Duplication d'une liste existante*

Spécification de l'algorithme

```
FONCTION duplicate(l : list) : list
VAR tête, copie, copie_tête : list
DEBUT
  SI (l = NULL) ALORS
    RETOURNER NULL
  FINSI
  tête ← l
  copie ← newSingletonList(l→content)
  copie_tête ← copie
  TANTQUE (l→succ ≠ NULL) ET (l→succ ≠ tête) FAIRE
    l ← l→succ;    // parcours de la liste d'origine
    // création d'une place à partir de la place d'origine
    copie→succ ← newSingletonList(l→content)
    // insertion de la nouvelle place dans la copie
    SI (l→prev ≠ NULL) ALORS
      copie→succ→prev ← copie
    FINSI
    copie ← copie→succ
  FAIT
  SI (l→succ ≠ NULL) ALORS
    copie→succ ← copie_tête
  FINSI
  RETOURNER copie_tête
FIN
```

Implantation C

```
list duplicate(list l)
{
  if (l == NULL) return NULL;
  list head = l;
  list dupl = newSingletonList(l->content);
  list dupl_head = dupl;
  while (l->succ != NULL && l->succ != head)
  {
    l = l->succ;    // itération de la liste de référence
    dupl->succ = newSingletonList(l->content);
    if (l->prev != NULL) dupl->succ->prev = dupl;
    dupl = dupl->succ; // itération de la liste copiée
  }
```

```
  if (l->succ != NULL) dupl->succ = dupl_head;
  return dupl_head;
}
```

Exercice 2.8 Refermer une liste sur elle-même

Spécification

```
Donnée modifiée : la liste transformée
FONCTION refermer_sur_elle_meme(*l: liste)
VAR p: place
DEBUT
  SI estvide(*l) ALORS
    RETOURNER
  FINSI
  p ← tête(*l)
  TANTQUE ¬ dernier(p) FAIRE
    p ← succ(p)
  FAIT
  succ(p) ← tête(*l)
FIN
```

Réalisation en C

```
Donnée modifiée : la liste transformée
void lin2circ(plist pl)
{
  if (pl == NULL) return;
  list l = *pl;
  if (l == NULL) return;
  while (l->succ != NULL) l = l->succ;  // itération de la liste
  l->succ = *pl;
}
```

Exercice 2.9 Effectuer et retourner deux calculs sur une liste

Étude

Il s'agit de parcourir la liste et de faire le produit d'une part les entiers positifs et d'autre part celui des entiers négatifs.

La présence éventuelle du 0 qui selon le statut qu'on lui donne est un positif ou non (positifs strictement ou non) peut poser un problème d'interprétation de l'énoncé. Le minimum est d'en parler. Disons, pour ce corrigé qu'il n'est ni positif ni négatif et qu'il doit donc ne pas intervenir dans aucun produit qu'il annulerait.

Les cas spéciaux sont les suivants :

- Liste vide ou ne contenant que des zéros : le résultat est indéterminé.
- Absence de positifs ou (exclusif) absence de négatifs : le résultat est partiellement indéterminé.

La difficulté de cet exercice tient également au problème du double résultat qu'il s'agit de retourner. Plusieurs solutions sont possibles, la plus simple consistant à utiliser deux entrées modifiées.

On utilisera un code de statut pour rendre compte des résultats spéciaux vs. standard dont les valeurs signifient les configurations suivantes :

- 4 : au moins un positif et au moins un négatif.
- 2 : au moins un positif mais aucun négatif.
- 1 : au moins un négatif mais aucun positif.
- 0 : ni négatif, ni positif : liste vide ou de zéros.

Spécification abstraite

```
Entrée : la liste d'entiers.
Entrée modifiée : le produit des positifs et le produit des négatifs.
Sortie : code de statut qui peut prendre l'un des 4 états TOUT (4),
UNIQUEMENT_POSITIFS (2), UNIQUEMENT_NEGATIFS (1), RIEN (0).
Algorithme :
FONCTION produits(l: liste<entier>, pos, neg: entier): statut
VAR pos_ok, neg_ok: booléen; p: place;
DEBUT
  pos ← 1
  pos_ok ← faux
  neg ← 1
  neg_ok ← faux
  SI ¬ estvide(l) ALORS
    p ← tête(l)
    SI contenu(p) > 0 ALORS
      pos ← contenu(p)
      pos_ok ← vrai
    FINSI
    SI contenu(p) < 0 ALORS
      neg ← contenu(p)
      neg_ok ← vrai
    FINSI
    TANTQUE ¬ dernier(p) FAIRE
      p ← succ(p)
      SI contenu(p) > 0 ALORS
        pos ← pos * contenu(p)
        pos_ok ← vrai
      FINSI
      SI contenu(p) < 0 ALORS
        neg ← neg * contenu(p)
        neg_ok ← vrai
      FINSI
    FAIT
```

```
  FINSI
  SI pos_ok ALORS
     SI neg_ok ALORS RETOURNER TOUT
    SINON RETOURNER UNIQUEMENT_POSITIFS
    FINSI
  SINON
    SI neg_ok ALORS RETOURNER UNIQUEMENT_NEGATIFS
    SINON RETOURNER RIEN
    FINSI
  FINSI
FIN
```

Implantation C

```
// status code : 4 : tout, 2 : positifs, 1 : négatifs, 0 : rien
int produits(list l, int *pos, int *neg)
{
  if (l == NULL) return 0;
  int pos_prod = 1;
  int pos_prod_ok = 0;
  int neg_prod = 1;
  int neg_prod_ok = 0;
  int content = l->content;  // pas indispensable mais économique
  if (content > 0)
  {
    pos_prod = content;
    pos_prod_ok = 1;
  }
  if (content < 0)
  {
    neg_prod = content;
    neg_prod_ok = 1;
  }
  while (l->succ != NULL)
  {
    l = l->succ;
    content = l->content;
    if (contenu > 0)
    {
      pos_prod *= content;
      pos_prod_ok = 1;
    }
    if (contenu < 0)
    {
```

```
            neg_prod *= content;
            neg_prod_ok = 1;
          }
        }
        if (pos_prod_ok) *pos = pos_prod;
        if (neg_prod_ok) *neg = neg_prod;
        if (pos_prod_ok && neg_prod_ok) return 4;
        if (pos_prod_ok && ! neg_prod_ok) return 2;
        if (! pos_prod_ok && neg_prod_ok) return 1;
        return 0;
      }
```

Exercice 2.10 Couper une liste en deux

Étude

Dans les cinq cas de figure, il s'agit de vérifier l'existence d'une position de coupe dans la chaîne, et le cas échéant d'effectuer celle-ci, puis de savoir retourner les deux chaînes.
Le cas à évacuer en début de fonction est celui de la liste vide.
Ensuite, nous devons, respectivement pour les cinq cas :

- A – vérifier que le pointeur désigne bien un maillon de la liste ;
- B – vérifier que la liste contient bien la donnée *x* ;
- C – vérifier que la liste contient au moins *k* éléments ;
- D – identifier la valeur minimale de la liste ;
- E – mesurer la longueur de la liste.

Les cas A, B et C sont très similaires et doivent intégrer les cas de tête et de queue de liste.
Les cas D, E nécessitent un premier parcours complet de la liste.
Tous les cas sauf le cas E doivent intégrer la possibilité d'une coupure devant la tête de liste.
Le cas E revient quasiment au cas B après l'identification du minimum de la liste.
Dans tous les cas, il est pertinent de retourner un statut d'exécution qui indique si l'opération a été effective ou bien transparente (auquel cas, il n'y a pas eu production d'une seconde liste).
Il peut y avoir erreur si l'un ou l'autre des deux pointeurs sur les listes à modifier est NULL ou bien dans le cas particulier du A (B et C sont discutables) si le pointeur donné pointe sur une autre liste.
Comme nous sommes scrupuleux, nous indiquons également par le statut d'exécution si la seconde liste n'était pas initialisée à NULL avant l'opération effective (mise en garde : écrasement d'une ancienne valeur).

Implantation C

Schéma commun

Données modifiées : l_1, la liste d'entrée coupée en deux qui devient la liste amont, et l_2 la liste aval éventuellement produite en sortie.
Résulat : un entier comme statut d'exécution
  -1 = ERR :ERREUR,

```
   0       : opération blanche,
   1       : une coupe a été effectuée,
   2       : statut 1 + l2 n'était pas nulle.
int cut<Version>(plist pl1, plist pl2, ...)
{
  if (pl1 == NULL || pl2 == NULL) return ERR;
  list l1 = *pl1;
  if (l1 == NULL) return 0;
  // on fixe le statut par défaut selon que *pl2 est NULL ou non
  int status = *pl2 == NULL ? 1 : 2;
  /// CODE SPECIFIQUE <Version> ///
  // on coupe
  *pl2 = l1->succ;
  l1->succ = NULL;
  return status;
}
```

Cas du sélecteur « pointeur »

```
Donnée : cutPoint, le pointeur sur la cellule suivant le point de coupe
int cutA(plist pl1, plist pl2, list cutPoint)
{
  /// CODE GENERIQUE COMMUN D'INITIALISATION ///
  // si cutPoint est NULL, autant arrêter ici
  if (cutPoint == NULL) return 0;
  // cas de la coupure en tête de liste :
  if (cutPoint == l1)
  {
    *pl1 = NULL;
    *pl2 = l1;
    return status;
  }
  // sinon, on parcourt la liste jusqu'au prédécesseur du point de coupe
  // ou jusqu'à la queue de liste (cutPoint pointe sur une autre liste)
  while (l1->succ != NULL && l1->succ != cutPoint) l1 = l1->succ;
  // si nous avons atteint la fin de liste, c'est un cas d'erreur
  if (l1->succ == NULL) return ERR;
  /// CODE GENERIQUE COMMUN DE FINALISATION ///
}
```

Cas du sélecteur « élément »

```
Donnée : x, l'élément de la cellule suivant le point de coupe
int cutB(plist pl1, plist pl2, int x)
{
```

```
  /// CODE GENERIQUE COMMUN D'INITIALISATION ///
  // cas de la coupure en tête de liste :
  if (x == l1->content)
  {
    *pl1 = NULL;
    *pl2 = l1;
    return status;
  }
  // sinon, on parcourt la liste jusqu'au prédécesseur du point de coupe
  // ou jusqu'à la queue de liste (pas d'élément x dans la liste)
  while (l1->succ != NULL && l1->succ->content != x) l1 = l1->succ;
  // si nous avons atteint la fin de liste, c'est une opération blanche
  if (l1->succ == NULL) return 0;
  /// CODE GENERIQUE COMMUN DE FINALISATION ///
}
```

Cas du sélecteur « position »

Donnée : k, la distance de la tête de liste au point de coupe

```
int cutC(plist pl1, plist pl2, unsigned k)
{
  /// CODE GENERIQUE COMMUN D'INITIALISATION ///
  // cas de la coupure en tête de liste :
  if (k == 0)
  {
    *pl1 = NULL;
    *pl2 = l1;
    return status;
  }
  // sinon, on parcourt la liste jusqu'au prédécesseur du point de coupe
  // ou jusqu'à la queue de liste (cutPoint pointe sur une autre liste)
  while (l1->succ != NULL && --k > 0) l1 = l1->succ;
  // si nous avons atteint la fin de liste, c'est une erreur
  if (l1->succ == NULL) return ERR;
  /// CODE GENERIQUE COMMUN DE FINALISATION ///
}
```

Cas du sélecteur « élément minimal »

```
int cutD(plist pl1, plist pl2)
{
  /// CODE GENERIQUE COMMUN D'INITIALISATION ///
  // on réutilise la fonction minOf de mesure de longueur de liste
  int min = minOf(l1);  // sur vos copies, il faut la réécrire
  // cas de la coupure en tête de liste :
```

```
    if (min == l1->content)
    {
      *pl1 = NULL;
      *pl2 = l1;
      return status;
    }
    // sinon, on parcourt la liste jusqu'au prédécesseur du point de coupe
    while (l1->succ != NULL && l1->succ->content != min) l1 = l1->succ;
    // avoir atteint la queue de liste ne peut logiquement pas arriver..
    /// CODE GENERIQUE COMMUN DE FINALISATION ///
}
```

Cas du sélecteur « milieu de liste »

```
int cutE(plist pl1, plist pl2)
{
    /// CODE GENERIQUE COMMUN D'INITIALISATION ///
    // on réutilise la fonction len de mesure de longueur de liste
    int n = len(l1);  // sur vos copies, il faut la réécrire
    // on calcule la distance au point de coupe (note: distance >= 1)
    int cutPoint = (n % 2 == 0) ? n / 2 : n / 2 + 1;
    int status = 1;
    // on modifie le statut si *pl2 n'est pas NULL
    if (*pl2 != NULL) status = 2;
    // on effectue le parcours jusqu'au prédécesseur du point de coupe
    while (--cutPoint > 0) l1 = l1->succ;
    // avoir atteint la queue de liste ne peut logiquement pas arriver..
    /// CODE GENERIQUE COMMUN DE FINALISATION ///
}
```

Spécification abstraite

Une fois n'est pas coutume (petit mensonge d'ingénieur), nous procédons à une rétro conception du principe algorithmique général à partir des cinq cas étudiés et réalisés précédemment.

Données modifiées : l_1, la liste d'entrée coupée en deux qui devient la liste amont, et l_2 la liste aval éventuellement produite en sortie.
Donnée : c, un paramètre du critère pour repérer la place qui suit directement le point de coupe et qui devient donc la tête de l_2
Résultat : un statut d'exécution

```
  -1 = ERR  :ERREUR,
   0 = ID : opération blanche,
   1 = OK : une coupe a été effectuée,
FONCTION scinder(modif l1, l2 = ø: liste; c: critère): statut
VAR p: place
```

```
DEBUT
  c ← mettre_à_jour(c, l1)
  SI ¬ estvalide(c) ALORS
    RETOURNER ERR
  FINSI
  SI estvide(l1) ALORS
    RETOURNER ID
  FINSI
  p ← tête(l1)
  SI est_successeur_point_de_coupe(p) ALORS
    l2 ← l1
    l1 ← ø
    RETOURNER OK
  FINSI
  TANTQUE ¬ dernier(p) ∧ ¬ est_successeur_point_de_coupe(succ(p)) FAIRE
    p ← succ(p)
    c ← mettre_éventuellement_à_jour(c)
  FAIT
  SI dernier(p) ALORS
    RETOURNER ID
  SINON
    tête(l2) ← succ(p)
    succ(p) ← ø
    RETOURNER OK
  FINSI
FIN
```

Exercice 2.11 Supprimer un sous-ensemble d'une liste

Étude

Cet exercice avec ses cinq cas de figure est une extension du principe algorithmique vu dans l'exercice 2.5 (supprimer des éléments). La façon de définir le critère de sélection des nœuds à supprimer change selon les cas, mais le schéma général ne change pas.

Dans tous les cas, il s'agit de traiter spécialement la chaîne préfixe (de tête) constitué d'une succession de nœuds à éliminer, puis s'il reste au moins un élément, d'éliminer tous les éléments répondant au critère et situés au-delà de la tête.

Le cas A donne lieu à une version simplifiée de l'algorithme général, laquelle profite à plein de la régularité des intervalles de suppression.

- *Schéma général pour les cas B à E*



La méthode générale de résolution reste la même, seul le critère de choix des éléments à supprimer change. Pour illustrer ce fait, dans la partie ***Spécification de l'algorithme***, le second paramètre de la fonction `removeAll` est en réalité une fonction nommée `critère`. Cette fonction `critère` a pour paramètre un `entier` (la valeur extraite de la liste) et pour sortie un `booléen` indiquant si la valeur

répond au critère de suppression. Cette fonction `critère` peut avoir d'autres paramètres, notamment un `entier` indiquant le seuil à partir duquel effectuer la suppression pour les cas D et E.

Spécification de l'algorithme

```
FONCTION removeAll(*pl:list, critère(entier,...) : booléen ): entier
VAR l, suppr : list
DEBUT
  SI (pl = NULL) ALORS
    RETOURNER -1 // -1 signale une erreur
  FINSI
  l ← *pl
  SI (l = NULL) ALORS
    RETOURNER 0 // 0 indique qu'aucune suppression n'est faite
  FINSI
  // suppression à partir de la tête de liste
  TANTQUE (l ≠ NULL) ET (critère(l→content,...)=vrai) FAIRE
    suppr ← l
    l ← suppr→succ
    LIBERER(suppr)
  FAIT
  *pl ← l
  // suite de la liste
  TANTQUE (l ≠ NULL) FAIRE
    TANTQUE (l→succ ≠ NULL) ET (critère(l→succ→content,...)=vrai) FAIRE
      suppr ← l→succ
      l→succ ← suppr →succ
      LIBERER(suppr)
    FAIT
    l ← l→succ
  FAIRE
  RETOURNER 1 // indique que tout s'est bien passé
FIN
```

Implantation C

```
Données modifiées : la liste dont on supprime des éléments.
Résultat : un entier comme statut d'exécution
  -1 = ERR : erreur,
   0 = ID  : opération blanche,
   1 = OK  : au moins une suppression a été effectuée.
int removeAll<V>(plist pl[, param<V>])
{
  if (pl == NULL) return ERR;  /// cas d'erreur
  list l = *pl;
```

```
    if (l == NULL) return ID;   /// cas d'opération blanche
    list killed;
    /// cas de suppression de toute la chaîne préfixe d'éléments impairs
    while (l != NULL && tosuppr<V>(l->content, param<V>))
    {
      killed = l;
      l = killed->succ;
      free(killed);
    }
    *pl = l;
    /// à partir de ce point : chaîne vide, soit tête non supprimée
    /// supression des éléments cf. critère et au-delà de la tête
    while (l != NULL)
    {
      while (l->succ != NULL && tosuppr<V>(l->succ->content, param<V>))
      {
        killed = l->succ;
        l->succ = killed->succ;
        free(killed);
      }
      /// on passe soit au conservé suivant, soit à NULL (fin de chaîne)
      l = l->succ;
    }
    return OK;
  }
```

- *Adaptations pour les cas B à E*

Les correspondances suivantes permettent de spécialiser le schéma général pour répondre au quatre cas de figure B, C, D, E :

Spécification de l'algorithme

```
FONCTION critère_estPair(n : entier) : booléen // casB
VAR pair : booléen
DEBUT
  SI n % 2 = 0 ALORS
    pair ← vrai
  SINON
    pair ← faux
  FINSI
  RETOURNER pair
FIN
FONCTION critère_estImpair(n : entier) : booléen // casC
DEBUT
  RETOURNER ¬ critère_estPair(n)
```

```
FIN
FONCTION critère_estSup(n : entier, seuil : entier) : booléen // casD
VAR estSup : booléen
DEBUT
  SI n > seuil ALORS
    estSup ← vrai
  SINON
    estSup ← faux
  FINSI
  RETOURNER estSup
FIN
FONCTION critère_estInf(n : entier, seuil : entier) : booléen // casE
VAR estInf : booléen
DEBUT
  estInf ← faux
  SI (n < seuil) ALORS
    estInf ← vrai
  FINSI
  RETOURNER estInf
FIN
```

Implantation C

```
int removeAllB(plist pl)
  param<B> ≡ ø
  tosuppr<B>(element) ≡ element % 2 == 0
int removeAllC(plist pl)
  param<C> ≡ ø
  tosuppr<C>(element) ≡ element % 2 == 1
int removeAllD(plist pl, int threshold)
  param<D> ≡ int threshold
  tosuppr<D>(element, threshold) ≡ element > threshold
int removeAllE(plist pl, int threshold)
  param<E> ≡ int threshold
  tosuppr<E>(element, threshold) ≡ element < threshold
```

- *Adaptations pour le cas A*

Le cas de figure A, du fait de la régularité des intervalles de suppression conduit à une version simplifiée du schéma général précédent. On élimine notamment un niveau d'imbrication de boucles *tantque*, traduction du fait qu'il n'existe pas de chaîne de plusieurs éléments directement successifs à supprimer.

Spécification de l'algorithme

```
FONCTION removeAllA(*pl : list) : entier
VAR l, suppr : list
DEBUT
  SI (pl = NULL) ALORS
    RETOURNER -1
  FINSI
  l ← *pl
  SI (l = NULL) ALORS
    RETOURNER 0
  FINSI
  *pl ← l→succ
  LIBERER(l)
  l ← *pl
  TANTQUE (l ≠ NULL) ET (l→succ ≠ NULL) FAIRE
    suppr ← l→succ
    l→succ ← suppr →succ
    LIBERER(suppr)
    l ← l→succ
  FAIT
  RETOURNER 1
FIN
```

Implantation C

```c
int removeAllA(plist pl)
{
  if (pl == NULL) return ERR;  /// cas d'erreur
  list l = *pl;
  if (l == NULL) return ID;   /// cas d'opération blanche
  /// suppression de la tête
  *pl = l->succ;
  free(l);
  l = *pl;
  /// supression à raison d'un nœud sur deux
  list killed;
  while (l != NULL && l->succ != NULL)
  {
    killed = l->succ;
    l->succ = killed->succ;
    free(killed);
    l = l->succ;
```

```
    }
    return OK;
}
```

Corrigé du problème

Problème 2.1 Saisir, enregistrer puis évaluer un polynôme

Étude

Pour le module de saisie, on s'inspirera de la fonctionnalité de construction de liste à partir de la saisie sur l'entrée standard, dont on fera une adaptation.

Pour l'évaluation, il s'agit d'un calcul itératif.

Deux fonctionnalités utilitaires pourront constituer une amélioration de la version de base :

- Une fonction d'affichage de la formule polynôme.
- Une fonction (laissée en exercice non corrigé) de normalisation du polynôme après sa saisie et avant son emploi pour des évaluations (optimisation).

- *La structure de donnée*

Pseudo code formel

```
STRUCTURE terme
  coeff    : réel
  exposant : entier
  *suivant : terme
TYPE *polynome : terme
```

Implantation C

```
typedef struct term
{
  long int coeff;   // intervalle : -2 147 483 648 à 2 147 483 647
  long int expon;
  struct term* nextTerm;
} term;
typedef term *polynomial;
```

- *Saisie du polynôme*

Implantation C

```
// Fonction dérivée de la fonction
// constr::newListFromStandardInput(list* pl) : int
void recordPolynomial(polynomial* pp)
{
  *pp = NULL;
  int size = 1;
  char coeffStr[11];
```

```
  char exponStr[11];
  printf("Saisissez une serie de coefficient et exposants,\n");
  printf("puis terminez votre saisie par le coeff 'fin' :\n");
  printf("coeff. %d> ", size);
  scanf("%s", coeffStr);
  if (strcmp("fin", coeffStr) == 0) return;
  printf("expon. %d> ", size);
  scanf("%s", exponStr);
  size++;
  *pp = malloc(sizeof(term));  // allocation et rattachement tête
  polynomial p = *pp;
  p->coeff = atol(coeffStr);
  p->expon = atol(exponStr);
  p->nextTerm = NULL;
  printf("coeff. %d> ", size);
  scanf("%s", coeffStr);
  if (strcmp("fin", coeffStr) == 0) return;
  polynomial prev_p;
  while (strcmp("fin", coeffStr) != 0)
  {
    prev_p = p;
    p = p->nextTerm;          // itération
    printf("expon. %d> ", size);
    scanf("%s", exponStr);
    size++;
    p = malloc(sizeof(term));    // allocation
    p->coeff = atol(coeffStr);
    p->expon = atol(exponStr);
    // important, car la valeur par défaut n'est pas forcément NULL !!!
    prev_p->nextTerm = p;
    printf("coeff. %d> ", size);
    scanf("%s", coeffStr);
  }
  p->nextTerm = NULL;
}
```

- *Normalisation du polynôme saisi*

Spécification de l'algorithme

```
FONCTION normalizePolynomial(*pp : polynome)
VAR ???
DEBUT
    ???
FIN
```

Implantation C

```
void normalizePolynomial(polynomial* pp)
{
     A VOUS DE JOUER !!
}
```

- *Évaluation du polynôme (instanciation)*

Spécification de l'algorithme

```
FONCTION evalPolynomial(p : polynome, x : réel) : réel
VAR résultat, valeur : réel
DEBUT
  résultat ← 0
  SI (p = NULL) ALORS
    RETOURNER résultat
  FINSI
  résultat ← résultat + (p→coeff) * (puissance(x, p→exposant))
  TANTQUE (p→suivant ≠ NULL) FAIRE
    p ← p→suivant
    valeur ← (p→coeff) * (puissance(x, p→exposant))
    résultat ← résultat+valeur
  FAIT
  RETOURNER résultat
FIN
```

Implantation C

```
long evalPolynomial(polynomial p, long x)
{
  long res = 0;
  if (p == NULL) return res;
  res += (p->coeff) * (powl(x, p->expon));
  printf("first term eval : %ld\n", res);
  while (p->nextTerm != NULL)
  {
    p = p->nextTerm;
    long termVal = (p->coeff) * (powl(x, p->expon));
    printf("next term eval : %ld\n", termVal);
    res += termVal;
  }
  return res;
}
```

- *Affichage du polynôme*

Spécification de l'algorithme

```
FONCTION printPolynomial(p : polynome)
DEBUT
  SI (p = NULL) ALORS
    RETOURNER
  FINSI
  printTerm(p)
  TANTQUE (p→suivant ≠ NULL) FAIRE
    p ← p→suivant
    printTerm(p)
  FAIT
FIN
```

Implantation C

```
void printPolynomial(polynomial p)
{
  if (p == NULL) return;
  printf("[");
  printTerm(p);
  while (p->nextTerm != NULL)
  {
    p = p->nextTerm;
    printTerm(p);
  }
  printf("]");
}
```

- *Affichage d'un terme du polynôme*

Spécification de l'algorithme

```
FONCTION printTerm(p : polynome)
DEBUT
  SI (p = NULL) ALORS
    RETOURNER
  FINSI
  SI (p→coeff = 0) ALORS
    RETOURNER
  FINSI
  SI (p→coeff = 1) ALORS
    SI (p→exposant = 0) ALORS
      AFFICHER("+1")
    SINON
```

```
      SI (p→exposant = 1) ALORS
        AFFICHER("+x")
      SINON
        AFFICHER("+x^", p→exposant)
      FINSI
    FINSI
    RETOURNER
  FINSI
  SI (p→coeff = -1) ALORS
    SI (p→exposant = 0) ALORS
      AFFICHER("-1")
    SINON
      SI (p→exposant = 1) ALORS
        AFFICHER("-x")
      SINON
        AFFICHER("-x^", p→exposant)
      FINSI
    RETOURNER
  FINSI
  SI (p→coeff < 0) ALORS
    SI (p→exposant = 0) ALORS
      AFFICHER(p→coeff)
    SINON
      SI (p→exposant = 1) ALORS
        AFFICHER(p→coeff)
      SINON
        AFFICHER(p→coeff, "x^", p→exposant)
      FINSI
    FINSI
  SINON
    SI (p→exposant = 0) ALORS
      AFFICHER("+", p→coeff)
    SINON
      SI (p→exposant = 1) ALORS
        AFFICHER("+", p→coeff, "x")
      SINON
        AFFICHER("+", p→coeff, "x^", p→exposant)
      FINSI
    FINSI
  FINSI
FIN
```

Implantation C

```
void printTerm(polynomial p)
{
  if (p == NULL) return;
  if (p->coeff == 0) return;
  if (p->coeff == 1) {
    if (p->expon == 0) printf("+1");
    else if (p->expon == 1) printf("+X");
    else printf("+X^(%ld)", p->expon);
    return;
  }
  if (p->coeff == -1) {
    if (p->expon == 0) printf("-1");
    else if (p->expon == 1) printf("-X");
    else printf("-X^(%ld)", p->expon);
    return;
  }
  if (p->coeff < 0)
  {
    if (p->expon == 0) printf("%ld", p->coeff);
    else if (p->expon == 1) printf("%ldX", p->coeff);
    else printf("%ldX^(%ld)", p->coeff, p->expon);
  }
  else
  {
    if (p->expon == 0) printf("+%ld", p->coeff);
    else if (p->expon == 1) printf("+%ldX", p->coeff);
    else printf("+%ldX^(%ld)", p->coeff, p->expon);
  }
}
```

- *Fonction principale (saisie et évaluations de polynômes)*

```
void testPolynomial()
{
  printf("\n\t\t>> TD 1.10 Saisie et evaluation de polynomes <<\n\n");
  int i = 1;
  char input[10];
  polynomial newPolynomial;
  printf("\nSouhaitez-vous creer un nouveau polynome depuis
            la ligne de commande (OUI/*) ?\n\n");
  printf("newPolynomial %d> ", i);
  scanf("%s", input);
  printf("\n");
```

```
    while (strcmp("OUI", input) == 0)
    {
      recordPolynomial(&newPolynomial);
      printf("Le polynome saisi est : ");
      printPolynomial(newPolynomial);
      printf("\n");
      testEvalPolynomial(newPolynomial);
      printf("\nSouhaitez-vous creer un nouveau polynome depuis
                la ligne de commande (OUI/*) ?\n\n");
      printf("newPolynomial %d> ", i);
      scanf("%s", input);
      printf("\n");
    }
}
```

- *Fonction principale déléguée (évaluations d'un polynôme)*

```
void testEvalPolynomial(polynomial p)
{
    int i = 1;
    char input[10];
    long x;
    printf("\nSouhaitez-vous evaluer le polynome pour une nouvelle
              valeur de x (OUI/*) ?\n\n");
    printf("eval %d> ", i);
    scanf("%s", input);
    printf("\n");
    while (strcmp("OUI", input) == 0)
    {
      printf("x %d> ", i);
      scanf("%s", input);
      printf("\n");
      x = atol(input);
      long result = evalPolynomial(p, x);
      printPolynomial(p);
      printf("(x = %ld) -> [%ld]\n", x, result);
      i++;
      printf("\nSouhaitez-vous evaluer le polynome pour une nouvelle
                valeur de x (OUI/*) ?\n\n");
      printf("eval %d> ", i);
      scanf("%s", input);
      printf("\n");
    }
}
```

3 STRUCTURES SÉQUENTIELLES COMPLEXES

RAPPELS DE COURS

3.1 PILES

Pour beaucoup d'applications, les seules opérations à effectuer sur les listes sont des insertions et des suppressions aux extrémités. Dans les **piles** les insertions et les suppressions se font à une seule extrémité, appelée **sommet de pile**.

Les piles sont aussi appelées **LIFO**, pour *Last-In-First-Out*, c'est-à-dire *dernier entré, premier sorti* ; une bonne image pour représenter une pile est une ... pile d'assiettes : c'est en haut de la pile qu'il faut prendre ou mettre une assiette !

Les opérations sur les piles sont : tester si une pile est vide, accéder au sommet d'une pile, empiler un élément, retirer l'élément qui se trouve au sommet (dépiler).

On peut utiliser pour implémenter les piles toutes les représentations possibles pour les listes. On va parler en détail d'une représentation chaînée, mais il faut comprendre que la représentation contiguë est tout à fait possible.

3.1.1 Représentation contiguë des piles

Les éléments de la pile sont rangés dans un tableau, et l'on conserve aussi l'indice du sommet de pile.

Définition d'une structure de pile réalisée avec un tableau

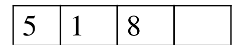
Pseudo code formel

```
STRUCTURE tStack
  content : T[n]
  top : entier
```

Implantation C

```
typedef struct tStack
{
  <type> content[n];
  int top;
} tStack;
```

La pile vide est représentée par tout enregistrement dont le champ `top` contient –1, car ce champ représente l'indice de la dernière case du tableau `content` de stockage des données.



5	1	8	

3.1.2 Représentation chaînée des piles

Les éléments de la pile sont chaînés entre eux, et le sommet d'une pile non vide est le pointeur vers le premier élément de la liste.

Exemple de définition d'une structure de pile chaînée

Pseudo code formel

```
STRUCTURE pile
  sommet : liste
  nb_elt : entier
```

Implantation C

```
typedef struct pile
{
  liste sommet;
  int nb_elt;
} pile;
```

3.1.3 Manipulation d'une pile

Initialiser une pile juste déclarée

Déclarations
Résultat de type `pile`
En-tête : `pile init()`

Algorithme « initialiser »

```
FONCTION init() : pile
VAR p : pile
DEBUT
  p.sommet ← NULL
  p.nb_elt ← 0
  RETOURNER p
FIN
```

Tester si la pile est vide

Déclarations
Donnée : `pile p`
Résultat : de type `booléen`
En-tête en C : `int pile_vide (pile p)`

Algorithme « pile vide »

```
FONCTION pilevide(p : pile) : booléen
VAR vide : booléen
DEBUT
  SI p.nb_elt=0 ALORS
    vide ← vrai
  SINON
    vide ← faux
  RETOURNER vide
FIN
```

Placer un élément au sommet de la pile (empiler)

Déclarations

Donnée : `T elt_à_emp`
Donnée modifiée : `pile *pp`
En-tête en C : `void empiler(T elt_a_emp, pile *pp);`

Algorithme « empiler »

```
FONCTION empiler(elt_a_emp : T ,*pp : pile)
VAR courant : liste
DEBUT
  RESERVER(courant)
  courant→info ← elt_à_emp
  // si la pile était vide alors sommet était NULL
  // et on crée la pile :
  courant→suivant ← pp→sommet
  pp→sommet ← courant  // rattache ancien sommet
  pp→nb_elt ← pp→nb_elt + 1
FIN
```

Retirer un élément du sommet de la pile (dépiler)

Déclarations

Donnée modifiée : `pile *pp, T *elt_dep`
Résultat : de type `booléen`
En-tête en C : `int depiler(pile *pp, T *elt_dep);`

Algorithme « dépiler »

```
FONCTION depiler(*pp :pile, *elt_dep : T) : booléen
VAR courant : ptr_poste; ok : booléen;
DEBUT
  SI non pile_vide(*pp) ALORS
    ok ← vrai
    *elt_dep ← pp→sommet→info
    pp→nb_elt ← pp→nb_elt -- 1
    // libération de l'espace mémoire :
    courant ← pp→sommet
    // si la pile contenait un seul élément,
```

```
    // elle devient vide, et *pp.sommet devient NULL
    pp→sommet ← pp→sommet→suivant
    LIBERER (courant)
  SINON
    ok ← faux
  RETOURNER ok
FIN
```

3.2 LES FILES

Dans le cas d'une **file** on fait les adjonctions à une extrémité, les accès et les suppressions à l'autre extrémité. Par analogie avec les files d'attente on dit que l'élément présent depuis le plus longtemps est le premier, on dit aussi qu'il est en tête.

Les files sont aussi appelées **FIFO** pour *First-In-First-Out*, c'est-à-dire *premier entré, premier sorti*.

Les opérations sur les files sont :

- tester si la file est vide ;
- accéder au premier élément de la file ;
- ajouter un élément dans la file ;
- retirer le premier élément de la file.

3.2.1 Représentation contiguë des files

Dans ce cas on doit conserver l'indice i du premier élément et l'indice j de la première case libre après la file. On fait progresser ces indices modulo la taille `lmax` du tableau. Le seul point délicat est la détection des débordements.

Définition d'une structure de file réalisée avec un tableau

Pseudo code formel

```
STRUCTURE file
  donnée : T[n]
  tête : entier
  fin : entier
```

Implantation en C

```
typedef struct file
{
  T donnée[n];
  int tête;
  int fin;
} file;
```

3.2.2 Représentation chaînée des files

Dans le cas d'une représentation chaînée, soit on a deux pointeurs, tête et dernier, vers le premier et le dernier élément de la file, soit on utilise le pointeur qui suit le dernier élément pour repérer le premier élément. On a donc une représentation circulaire.

3.2.3 Manipulation d'une file (méthode avec deux pointeurs)

Définition d'une structure de file chaînée

Pseudo code formel

```
STRUCTURE poste
  info : T
  *suivant : poste
TYPE *ptr_poste : poste
STRUCTURE file
  tête : ptr_poste
  fin : ptr_poste
  nb_elt : entier
```

Implantation en C

```
typedef struct poste
{
  T info;
  struct poste *suivant;
} poste;
typedef poste*  ptr_poste;
typedef struct file
{
  ptr_poste tête;
  ptr_poste fin;
  int nb_elt;
} file;
```

Initialiser une file juste déclarée

Déclarations
Résultat de type `file`
En-tête en C : `file init()`

Algorithme « initialiser »

```
FONCTION init()
VAR f : file
DEBUT
  f.tete ← NULL
  f.fin ← NULL
  f.nb_elt ← 0
  RETOURNER f
FIN
```

Tester si la file est vide

Déclarations
Donnée : file f
Résultat : de type booléen
En-tête en C : int file_vide (file f)

Algorithme file vide

```
FONCTION file_vide(f :file) : booléen
VAR vide: booléen
DEBUT
  SI f.nb_elt = 0 ALORS
    vide ← vrai
  SINON
    vide ← faux
  FINSI
  RETOURNER vide
FIN
```

Placer un élément à la fin de la file (enfiler)

Déclarations
Donnée : T elt_à_enf
Donnée modifiée : file *ff
En-tête C : void enfiler(T elt_a_enf, file *ff);

Algorithme « enfiler »

```
FONCTION enfiler(x : T, *ff : file)
VAR courant : ptr_poste
DEBUT
  RESERVER(courant)
  courant→info ← x
  courant→suivant ← NULL
  SI ff→tete = NULL ALORS
    ff→tete ← courant
    ff→fin ← courant
  SINON
    ff→fin→suivant ← courant
    ff→fin ← courant
  FINSI
  ff→nb_elt ← ff→nb_elt + 1
FIN
```

Retirer un élément de la tête de la file (défiler)

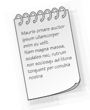
Déclarations
Donnée modifiée : file *ff, T *elt_def
Résultat : de type booléen
En-tête : int defiler(file* ff, T *elt_dep);

Algorithme « défiler »

```
FONCTION defiler(ff : file, *elt_def : T) : booléen
VAR courant : ptr_poste , ok : booléen
DEBUT
  SI non file_vide(*ff) ALORS
    ok ← vrai
    *elt_def ← ff→tete→info
    ff→nb_elt ← ff→nb_elt -- 1
    // opération pour libérer l'espace mémoire :
    courant ← ff→tete
    ff→tete ← ff→tete→suivant
    LIBERER(courant)
    SI ff→nb_elt = 0 ALORS
      // si la file contenait un seul élément,
      // elle devient vide, et ff→fin devient NULL,
      // ainsi que ff→tete
      ff→fin ← NULL
    FINSI
  SINON
    ok ← faux
  FINSI
  RETOURNER ok
FIN
```

Remarque

Une file se représente beaucoup mieux en dynamique (représentation chaînée).



Utilisation des files : tous les cas où l'on dispose d'une ressource destinée à être utilisée par plusieurs programmes, les cas où plusieurs données sont manipulées par cette ressource. Exemple : le buffer d'imprimante est géré en file d'attente.

Exemple de gestion d'un buffer clavier en liste circulaire (le dernier poste d'une liste circulaire est rattaché au premier)

On a un ensemble des données homogènes gérées sur un principe de lecture/écriture concurrente : lecture et écriture peuvent se produire de manière simultanée mais sont conditionnées par l'autre opération.

Une lecture ne peut être faite que si une écriture a déjà été faite, mais une écriture ne peut être faite que si la lecture a été effectuée sous peine de perdre l'information. Plus généralement : il s'agit d'une file d'attente particulière où l'on n'a pas besoin de produire toutes les données pour commencer à les consommer. Le nombre de postes est fixé au départ et ne varie plus au cours du programme.

Il faut écrire deux algorithmes qui utilisent respectivement un pointeur de lecture et un pointeur d'écriture :

- **Lire** : c'est prendre l'information référencée par lecteur, à condition que cela soit possible et déplacer le pointeur de lecture sur le poste suivant.

- **Écrire** : c'est mettre une information référencée par le pointeur `Ecriture` si cela est possible, et déplacer le pointeur d'écriture sur le poste suivant.

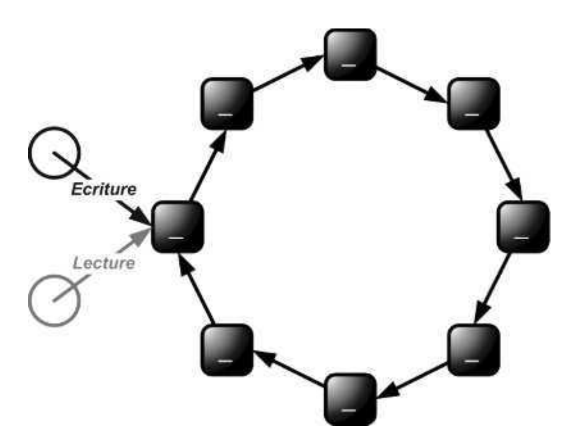


Figure 3.1 État initial du buffer clavier



On ne peut pas lire si rien n'a été écrit ou si l'information a été déjà lue. On ne peut pas écrire si le poste contient une information encore non lue.

Exemple

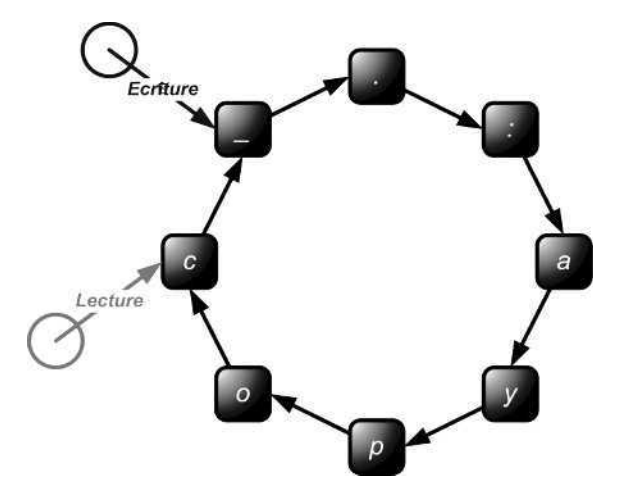


Figure 3.2 Écriture

`C\> copy a: .`
le fait de taper `RC` entame la lecture
On tape `dir` très rapidement

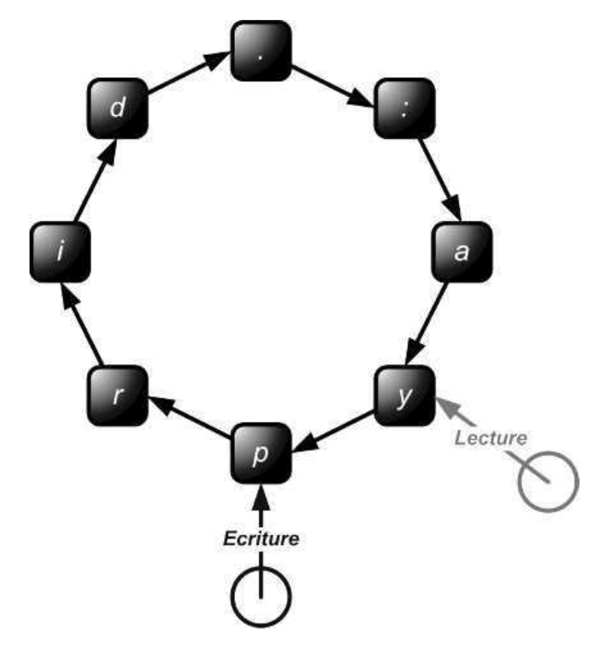


Figure 3.3 Lecture et écriture simultanées

Modèle statique

Pseudo code formel

```
STRUCTURE elt
  info : T
  àlire : booléen
```

Implantation C

```
typedef struct elt {
  T info;
  booléen àlire;
} elt;
```

àlire est un *flag* (drapeau) :
àlire = vrai : on peut lire, mais pas écrire ;
àlire = faux : on peut écrire, mais pas lire.

Pseudo code formel

```
SRUCTURE liste_circ
  donnée : elt[n]
  lecteur : entier
  écrivain : entier
```

Implantation C

```
typedef struct liste_circ {
  elt donnée [n];
  int lecteur;
  int écrivain;
} liste_circ
```

Initialisation

Déclarations
Donnée : int n
Donnée modifiée : liste_circ *l
En-tête : void init(liste_circ *l);

Algorithme

```
FONCTION init(*l : liste_circ)
VAR i : entier
DEBUT
  POUR i DE 0 A n-1 FAIRE
    l→donnée[i].àlire ← faux
    l→lecteur ← 0
    l→ecrivain ← 0
  FAIT
FIN
```

Lecture

Donnée : int n
Donnée modifiée : T *elt_à_lire, liste_circ *l
Résultat de type booléen
En-tête : int lecture(liste_circ *l, T *elt_à_lire)

```
FONCTION lecture(*l : liste_circ, *elt_à_lire : T)
VAR ok : booléen
DEBUT
  SI l→donnée[l→lecteur].àlire = faux ALORS
    ok ← faux
  SINON
    ok ← vrai
    *elt_à_lire ← l→donnée[l→lecteur].info
   l→donnée[l→lecteur].àlire ← faux
   l→lecteur ← (l→lecteur + 1) mod n
  FINSI
  RETOURNER ok
FIN
```

Écriture

Donnée : int n, T elt_ecr
Donnée modifiée : liste_circ *l
Résultat de type booléen
En-tête : int écriture(T elt_ecr, liste_circ *l);

```
FONCTION écriture(elt_ecr : T, *l : liste_circ) : booléen
VAR ok : booléen
DEBUT
  SI l→donnée[l→ecrivain].àlire = vrai ALORS
    ok ← faux
  SINON
    ok ← vrai
    l→donnée[l→ecrivain].info ← elt_ecr
    l→donnée[l→ecrivain].àlire ← vrai
    l→ecrivain ← (l→ecrivain + 1) mod n
  FINSI
  RETOURNER ok
FIN
```

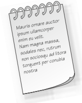
Modèle dynamique

Pseudo code formel

```
STRUCTURE elt
  info : T
  àlire : booléen
  *suivant : elt
TYPE *ptr_elt : elt
STRUCTURE liste_circ
  lire : ptr_elt
  ecrire : ptr_elt
```

Implantation C

```
typedef struct elt
{
  T info;
  booléen àlire;
  struct elt *suivant;
} elt;
typedef ptr_elt *elt;
typedef struct liste_circ {
  ptr_elt lire;
  ptr_elt ecrire;
} liste_circ;
```



Rien dans les types ne traduit que le premier élément de la liste est le successeur du dernier : ceci devra être rendu explicite dans les algorithmes.
En approche statique il s'agit de l'opération modulo N
En approche dynamique il s'agit du passage au suivant

ÉNONCÉS DES EXERCICES ET DES PROBLÈMES

EXERCICES

Exercice 3.1 Afficher une liste chaînée ****

Écrire un algorithme pour afficher une liste chaînée d'entiers sous la forme suivante :
Si la liste est linéaire :

```
Nom_liste: .->[elt1]->[elt2]->...->[eltn]->[eltlast] |
```

Si la liste est circulaire :

```
Nom_liste: .->[elt1]->[elt2]->...->[eltn]->[eltlast] ...
```

Si certains maillons contigus sont doublement chaînés, afficher `]<->[` au lieu de `]->[`.



Apportez une attention particulière aux traitements spécifiques de début, milieu et fin de liste ainsi qu'aux cas spéciaux comme la liste vide ou singleton.

Exercice 3.2 Construire une liste circulaire ordonnée ***

Écrire un algorithme qui crée une liste circulaire ordonnée d'entiers à partir d'un tableau non ordonné d'entiers.

Exercice 3.3 Réaliser le chaînage arrière d'une liste doublement chaînée *

Concevoir un algorithme qui réalise le chaînage arrière d'une liste doublement chaînée dont seul le chaînage avant a été effectué.

Exercice 3.4 Inverser pile et file **

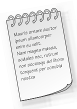
Concevoir deux algorithmes, qui créent respectivement :

- la file inverse d'une file ;
- la pile inverse d'une pile.

Ces deux algorithmes doivent restituer leur entrée inchangée.

Exercice 3.5 Simuler la récursivité à l'aide d'une pile *

Écrire un algorithme pour calculer la somme de 1 à n $\in n \in \mathbb{N}^*$ en simulant la récursivité à l'aide d'une pile.



Les en-têtes des opérations à utiliser pour cet exercice sont fournis :
FONCTION nouvellePile() : pile
FONCTION estPileVide(p : pile) : booléen
FONCTION empiler (val : T, *pp : pile)
FONCTION depiler (*pval : T, *pp : pile) : entier

Exercice 3.6 Insérer et supprimer dans une liste doublement chaînée **

Écrire un algorithme d'insertion dans une liste doublement chaînée.
Écrire un algorithme de suppression dans une liste doublement chaînée.

PROBLÈMES

Problème 3.1 Problème de Joseph ****

Écrire un algorithme permettant de lire deux entiers positifs n et k. Construire une liste circulaire dans laquelle seront enregistrés les nombres *1, 2, 3, ..., n*, dans cet ordre. En commençant à partir du nœud contenant *1*, supprimer successivement tous les $k^{ièmes}$ nœuds de la liste, en effectuant un parcours circulaire dans celle-ci et jusqu'à ce que tous les nœuds aient été supprimés. Dès qu'un nœud est supprimé, le suivant dans la boucle est considéré comme la nouvelle tête de liste et ainsi de suite. Si $n = 8$ et $k = 3$, par exemple, la suppression des nœuds contenant les huit entiers se fait dans cet ordre :

$$3, 6, 1, 5, 2, 8, 4, 7$$

(Ce procédé peut être illustré par un groupe composé à l'origine de n personnes formant un cercle, que l'on élimine successivement en désignant le $k^{ième}$ de ceux restant dans le cercle que l'on continue de parcourir).

Problème 3.2 Mesurer la longueur d'une liste de type Heqat *****

Soit une liste simplement chaînée de type Heqat, c'est-à-dire obtenue par l'opération `concat(&l, sublist(&l, n))`, concevoir un algorithme pour en mesurer la longueur en nombre de nœuds, qui retourne d'une part la longueur de la partie amont linéaire, et d'autre part la longueur (périmètre) de la partie circulaire en aval.

Vous devrez respecter les contraintes suivantes :

- pas de modification de la liste pendant le traitement (lecture seule) ;
- pas de copie de l'information des nœuds traversés.

Corrigés des exercices et des problèmes

PRÉAMBULE

Représentation d'une structure de file à l'aide d'une liste

Pseudo code formel

```
STRUCTURE lQueue
  head : list
  queue : list
  length : entier
```

```
// quelques fonctions utiles, on fournit leurs en-têtes
FONCTION newEmptyQueue() : lQueue*
FONCTION isEmptyQueue(*p_q : lQueue) : booléen
FONCTION qPush(e : entier, *p_q : lQueue)
FONCTION qPop(*pe : entier, *p_q : lQueue) : entier
```

Implantation en C

```
/** structure de file réalisée avec une liste simplement chaînée
dont la tête est l'entrée, la queue la sortie **/
typedef struct lQueue
{
  list head;
  list queue;
  int length;
} lQueue;
/** Opérations **/
lQueue *newEmptyQueue();
int isEmptyQueue(lQueue *p_q);
void qPush(int e, lQueue *p_q);
int qPop(int* pe, lQueue* p_q);
```

Représentation d'une structure de pile à l'aide d'une liste

Pseudo code formel

```
STRUCTURE lStack
  top : list
  depth : entier
// quelques fonctions utiles, on fournit leurs en-têtes
FONCTION newEmptyStack() : lStack*
FONCTION isEmptyStack(*p_s : lStack) : booléen
FONCTION sPush(e : entier, *p_s : lStack)
FONCTION sPop(*pe : entier, *p_s : lStack) : entier
```

Implantation en C

```
/** structure de pile réalisée avec une liste simplement chaînée
dont la tête est le sommet, la queue la base **/
typedef struct lStack
{
  list top;
  int depth;
} lStack;
/** Opérations **/
```

```
lStack *newEmptyStack();
int isEmptyStack(lStack *p_s);
void sPush(int e, lStack *p_s);
int sPop(int* pe, lStack* p_s);
```

CORRIGÉS DES EXERCICES

Exercice 3.1 Afficher une liste chaînée

Spécification de l'algorithme

```
FONCTION printListGraph(l : list, nom : caractere[])
VAR tête : list
DEBUT
  SI (l = NULL) ALORS
    AFFICHER(nom, ‘‘: .| la liste est vide’’)
  SINON
    tête ← l
    AFFICHER(nom," : .")
    SI (tête→prev = NULL) ALORS // début de liste ?
      AFFICHER("->",tête→content)
    SINON
      AFFICHER("<->",tête→content)
    FINSI
    TANTQUE (l→succ ≠ NULL) ET (l→succ ≠ tête) FAIRE
      l ← l→succ
      SI (l→prev = NULL) ALORS
        AFFICHER("->", l→content)
      SINON
        AFFICHER("<->", l→content)
      FINSI
    FAIT
    SI (l→succ = NULL) ALORS
      AFFICHER(" |")
    SINON
      AFFICHER(" ...")
    FINSI
  FINSI
FIN
```

Réalisation en C

```
void printListGraph(list l, char * nom)
{
  if (l == NULL) printf("%s : . | (empty list !!)", nom);
  else
```

```
    {
      list head = l;
      printf("%s : .", nom);
      if (head->prev == NULL) printf("->[%d]", head->content);
      else printf("<->[%d]", head->content);
      while (l->succ != NULL && l->succ != head)
      {
        l = l->succ;
        if (l->prev == NULL) printf("->[%d]", l->content);
        else printf("<->[%d]", l->content);
      }
      (l->succ == NULL) ? printf(" |") : printf(" ...");
    }
    printf("\n");
}
```

Exercice 3.2 Construire une liste circulaire ordonnée

Analyse

Dans cet exercice, la circularité de la liste à produire est accessoire. Il est naturel de penser à travailler sur une liste linéaire standard puis de finaliser en la refermant sur elle-même avec un `concat(&l, &l)`.

Il s'agit de réaliser un tri :

- Soit en triant avant ou après l'appel au constructeur (méthode en deux temps) :
  - soit tri du tableau avant d'appeler ce constructeur,
  - soit tri de la liste après l'avoir construite.
- Soit en triant au fur et à mesure de la construction de la liste (méthode en flux tendu) :
  - soit au moment de choisir le prochain élément à insérer en tête de liste (itération non linéaire sur le tableau),
  - soit au moment de choisir l'emplacement d'insertion dans la liste de l'élément suivant du tableau (itération linéaire du tableau).
- Avec une difficulté (mais quelle élégance) supplémentaire si on choisit d'insérer dans une liste circulaire dès son initialisation.

Le mode de tri n'était pas imposé : vous pouvez laisser libre cours à votre savoir-faire et vos connaissances en ce domaine.

Réalisation

Ci-après deux solutions (méthode en flux tendu), l'une itérative qui parcourt le tableau, et qui pour chaque étape crée puis insère un nouveau maillon dans la liste, puis se termine en refermant la chaîne sur elle-même, l'autre, plus compacte, récursive, qui travaille sur une liste circulaire initiée avec le premier élément du tableau.

- *Méthode itérative classique*

Spécification de l'algorithme

```
/** construit une liste circulaire ordonnée d'entiers
à partir d'un tableau non ordonné d'entiers **/
FONCTION newOrdoredCircList(données : entier[], size : entier) : list
VAR i, valeur: entier ; trouvé : booléen; tete, sl_val, l : list
DEBUT
  SI size=0 ALORS // pas de données -> liste vide
    RETOURNER NULL
  FINSI
  // construction de la tête à partir du tableau de données
  tete ← newSingletonList(données[0])
  l ← tete
  POUR i DE 1 A size-1 FAIRE
    valeur ← données[i]
    trouve ← faux
    sl_val ← newSingletonList(valeur)
    // insertion en tête de liste
    SI valeur < l→content ALORS
      sl_val→succ ← l
      l ← sl_val
      tete ← l
    SINON  // insertion à faire en milieu de liste
      TANTQUE (l→succ ≠ NULL) ET (trouvé = faux) FAIRE
        SI val < l→succ→content ALORS // il faut insérer ici
          sl_val→succ ← l→succ
          l→succ ← sl_val
          trouvé ← vrai
        SINON   // parcours de la liste : passage au suivant
          l ← l→succ
        FINSI
      FAIT
      SI l→succ = NULL ALORS // cas où on a atteint la fin de liste
        l→succ ← sl_val
      FINSI
    FINSI
  FAIT
  concat(&tete,&tete) // pour rendre la liste circulaire
  RETOURNER tete
FIN
```

Implantation C

```
/** construit une liste circulaire ordonnée d'entiers
à partir d'un tableau non ordonné d'entiers **/
list newOrdoredCircList(int *data, unsigned size)
{
  if (data == NULL) return NULL;
  int i;
  int val;
  int found;
  list head = newSingletonList(data[0]);
  list sl_val, l = head;
  for (i = 1; i < size; i++)
  {
    val = data[i];
    found = 0;
    sl_val = newSingletonList(val);
    /// insertion en tête de liste
    if (val < l->content)
    {
      sl_val->succ = l;
      l = sl_val;
      head = l;
    }
    else
    {
      /// insertion en milieu de liste
      while (l->succ != NULL && !found)
      {
        if (val < l->succ->content)
        {
          sl_val->succ = l->succ;
          l->succ = sl_val;
          found = 1;
        }
        else l = l->succ;
      }
      /// insertion en fin de liste
      if (l->succ == NULL) l->succ = sl_val;
    }
  }
  concat(&head, &head);
  return head;
}
```

• *Méthode récursive*

Spécification de l'algorithme

```
/** Construit une liste circulaire ordonnée d'entiers
à partir d'un tableau non ordonné d'entiers **/
FONCTION newOrdoredCircListRec(données : entier[], size : entier) : list
VAR valeur : entier ; trouvé : booléen ; l, tête, l_succ, liste0 : list
DEBUT
  SI size = 0 ALORS
    RETOURNER  NULL
  FINSI
  SI size = 1 ALORS
    liste0 ← newSingletonList(données[0])
    liste0→succ ← list0
    RETOURNER liste0
  FINSI
  // appel récursif, où données+1 représente le sous-tableau
  // du tableau données commençant à l'indice suivant
  // (on a  supprimé  la première case du tableau)
  l ← newOrdoredCircListRec(données+1,size-1)
  tête ← l
  valeur ← données[0]
  trouvé ← faux
  TANTQUE (l→succ ≠ tête) ET (trouvé = faux) FAIRE
    SI l→content > l→succ→content ALORS
      SI (val > l→content) OU ¬ (val > l→succ→content) ALORS
        trouvé ← vrai
      SINON
        l ← l→succ
      FINSI
    SINON
      SI (val > l→content) ET ¬ (val > l→succ→content) ALORS
        trouvé ← vrai
      SINON
        l ← l→succ
      FINSI
    FINSI
  FAIT
  l_succ ← l→succ
  l→succ ← newSingletonList(données[0])
  l→succ→succ ← l_succ
  l ← l→succ
  RETOURNER l
FIN
```

Implantation C

```
/** Construit une liste circulaire ordonnée d'entiers
à partir d'un tableau non ordonné d'entiers **/
list newOrdoredCircListRec(int *data, unsigned size)
{
  if (size == 0) return NULL;
  if (size == 1)
  {
    list l0 = newSingletonList(data[0]);
    l0->succ = l0;
    return l0;
  }
  list l = newOrdoredCircListRec(data + 1, size - 1);
  list l_head = l;
  int val = data[0];
  int found = 0;
  while (l->succ != l_head && !found)
  {
    if (l->content > l->succ->content)
    {
      if (val > l->content || !(val > l->succ->content)) found = 1;
      else l = l->succ;
    }
    else
    {
      if (val > l->content && !(val > l->succ->content)) found = 1;
      else l = l->succ;
    }
  }
  list l_succ = l->succ;
  l->succ = newSingletonList(data[0]);
  l->succ->succ = l_succ;
  l = l->succ;
  return l;
}
```

Remarque

On peut éliminer le recours à la variable `'found'` et mieux factoriser la boucle d'itération avec la forme logique équivalente suivante :

```
while
(
  !(l->succ == l_head)
  &&
```

```
      (
        !(val > l->content)
        ||
        (val > l->succ->content)
        ||
        (l->content > l->succ->content)
      )
      &&
      (
        (val > l->content)
        ||
        !(val > l->succ->content)
        ||
        !(l->content > l->succ->content)
      )
    ) l = l->succ;
```

• *Pour effectuer des tests*

```
void testOrderedCirc()
{
  printf(">> ordered circular lists\n");
  int data[12] = {29, 23, 17, 11, 5, 2, 3, 7, 13, 19, 25, 31};
  printListGraph(newOrdoredCircList[Rec](data, 12), "OCL ({29 ... 31})");
  printListGraph(newOrdoredCircList[Rec](NULL, 0), "OCL (NULL)");
  printListGraph(newOrdoredCircList[Rec]((int[]) {1}, 1), "OCL ({1})");
}
```

Exercice 3.3 Réaliser le chaînage arrière d'une liste doublement chaînée

Spécification de l'algorithme

```
FONCTION doubleChain(l : list)
VAR tête : list
DEBUT
  SI (l = NULL) ALORS
    RETOURNER // ceci termine la fonction
  FINSI
  tête ← l
  TANTQUE (l→succ ≠ NULL) ET (l→succ ≠ tête) FAIRE
    SI (l→succ→prev = NULL) ALORS
      l→succ→prev ← l
    FINSI
    l ← l→succ
  FAIT
  SI (l→succ ≠ NULL) ALORS
    tête→prev ← l
```

```
  FINSI
  RETOURNER
FIN
```

Implantation C

```
void doubleChain(list l)
{
  if (l == NULL) return;
  list head = l;
  while (l->succ != NULL && l->succ != head)
  {
    if (l->succ->prev == NULL) l->succ->prev = l;
    l = l->succ;
  }
  if (l->succ != NULL) head->prev = l;
}
```

Exercice 3.4 Inverser pile et file

- *Inversion de pile*

Spécification de l'algorithme

```
FONCTION reverseQueue(*p_q : Queue) : lQueue*
VAR valeur, longueur : entier; *inversée : lQueue; *temp : lStack
DEBUT
  SI (p_q = NULL) ALORS
    RETOURNER NULL
  FINSI
  longueur ← p_q→length
  inversée ← newEmptyQueue() // résultat de l'inversion
  temp ← newEmptyStack() // pile pour stockage des valeurs
  // principe : empiler tous les éléments de la file
  // puis les dépiler dans la file inverse
  TANTQUE (longueur > 0) FAIRE
    qPop(&valeur, p_q)
    sPush(valeur, temp)
    qPush(valeur, p_q)
    longueur ← longueur-1
  FAIT
  /// construction de la file inverse :
  longueur ← p_q→length
  TANTQUE (longueur > 0) FAIRE
    sPop(&val, temp)
    qPush(val, inversée)
    longueur ← longueur-1
```

```
  FAIT
  RETOURNER inversée
FIN
```

Implantation C

Entrée : la file source dont il s'agit de construire l'inverse.
Sortie : la file inverse.

```
/** Crée la file inverse de la file '*p_q' **/
lQueue *reverseQueue(/*const*/lQueue *p_q)
{
  if (p_q == NULL)
  {
    fprintf(stderr, "reverseQueue error: null queue !!");
    return NULL;
  }
  int val;
  int length = p_q->length;
  lQueue *p_rev_q = newEmptyQueue(); /// la file inverse retournée
  lStack *p_tmp_s = newEmptyStack(); /// la pile pour l'inversion
  while (length-- > 0)
  {
    qPop(&val, p_q);     /// défilement depuis la file d'entrée '*p_q'
    sPush(val, p_tmp_s); /// empilement dans la pile d'inversion
    qPush(val, p_q);     /// réenfile val : rotation de '*p_q'
  }
  /// construction de la file inverse :
  length = p_q->length;
  while (length-- > 0)
  {
    sPop(&val, p_tmp_s);
    qPush(val, p_rev_q);
  }
  free(p_tmp_s); /// Soyons diligents avec la mémoire !!
  return p_rev_q;
}
/** Exemple d'inversion de file **/
void testQueueReversing()
{
  printf("\n\nInversion d'une file :\n\n");
  int val = 10;
  lQueue *p_q = newEmptyQueue();
  do qPush(val, p_q); while (--val > 0);
  printListGraph(p_q->head, "input queue:");
```

```
  lQueue *p_rev_q = reverseQueue(p_q);
  printListGraph(p_rev_q->head, "reversed queue:");
  printListGraph(p_q->head, "unchanged input queue:");
}
```

- *Inversion de pile*

Spécification de l'algorithme

```
FONCTION reverseStack(*p_s : lStack) : lStack *
VAR valeur : entier ; *inversée, *temp : lStack
DEBUT
  SI (p_s = NULL) ALORS
    RETOURNER NULL
  FINSI
  inversée ← newEmptyStack()
  temp ← newEmptyStack()
  FAIRE
    sPop(&valeur, p_s)
    sPush(valeur, inversée)
    sPush(valeur, temp)
  TANTQUE (¬ isEmptyStack(p_s))
  FAIRE
    sPop(&valeur, temp)
    sPush(valeur, p_s)
  TANTQUE (¬ isEmptyStack(temp))
  RETOURNER p_rev_s
FIN
```

Implantation C

Entrée : la pile source dont il s'agit de construire l'inverse.
Sortie : la pile inverse.

```
/** Crée la pile inverse de la pile '*p_s' **/
lStack *reverseStack(lStack *p_s)
{
  if (p_s == NULL)
  {
    fprintf(stderr, "reverseStack error: null stack !!");
    return NULL;
  }
  int val;
  lStack *p_rev_s = newEmptyStack(); /// la pile inverse retournée
  lStack *p_tmp_s = newEmptyStack(); /// une pile pour restituer '*p_s'
  do
```

```
  {
    sPop(&val, p_s);      /// dépilement depuis la pile d'entrée '*p_s'
    sPush(val, p_rev_s); /// empilement dans la pile résultat
    sPush(val, p_tmp_s); /// : pour reconstruction de '*p_s'
  }
  while (! isEmptyStack(p_s));
  /// reconstruction de '*p_s' :
  do
  {
    sPop(&val, p_tmp_s);
    sPush(val, p_s);
  }
  while (! isEmptyStack(p_tmp_s));
  free(p_tmp_s); /// Soyons diligents avec la mémoire !!
  return p_rev_s;
}
void testStackReversing()
{
  printf("\n\nInversion d'une file :\n\n");
  int val = 10;
  lQueue *p_q = newEmptyQueue();
  do qPush(val, p_q); while (--val > 0);
  printListGraph(p_q->head, "input queue:");
  lQueue *p_rev_q = reverseQueue(p_q);
  printListGraph(p_rev_q->head, "reversed queue:");
  printListGraph(p_q->head, "unchanged input queue:");
}
```

Exercice 3.5 Simuler la récursivité à l'aide d'une pile

Spécification de l'algorithme

```
FONCTION triangularSum(n : entier) : entier
VAR valeur, résultat : entier ; p : pile
DEBUT
  SI (n = 0) ALORS
    RETOURNER 0
  FINSI
  p ← initPile()
  valeur ← n
  résultat ← 0
  FAIRE // les valeurs sont mises sur la pile (empilées)
    empiler(valeur, &p)
    valeur ← valeur-1
  TANTQUE (valeur > 0)
```

```
  FAIRE // puis les valeurs sont sorties de la pile (dépilées)
    depiler(&val, &p)
    résultat ← résultat+valeur
  TANTQUE (¬ estPileVide(p))
  RETOURNER res
FIN
```

Implantation C

```
int triangularSum(unsigned n)
{
  if (n == 0) return 0;
  pile p = initPile();
  int val = n;
  int res = 0;
  do empiler(val, &p);
  while (--val > 0);
  do
  {
    depiler(&val, &p);
    res += val;
  }
  while (! estPileVide(p));
  return res;
}
```

Exercice 3.6 Insérer et supprimer dans une liste doublement chaînée

Étude

On procède de la même manière que pour le cas d'une liste simplement chaînée, à ceci près que l'algorithme est plus simple.

En effet, le problème pour une liste simplement chaînée consiste à repérer le prédécesseur direct du point d'insertion ou d'élimination, et implique donc un parcours linéaire en repartant de la tête.

Dans le cas d'une liste doublement chaînée, on dispose d'un accès direct sur l'élément précédent de la chaîne, éliminant ainsi cette difficulté.

Pour le reste, il s'agit de mettre à jour les prédécesseurs en sus des successeurs.

Les problèmes d'insertion/suppression en début ou fin de listes, vs. en milieu de liste restent à peu près les mêmes qu'avec une liste simplement chaînée.

- *Insertion*

Spécification de l'algorithme

```
// Insertion dans une liste doublement chaînée
// pl : liste où s'effectue l'insertion
// place : place dans la liste (on insère juste devant)
```

```
// k : élément à insérer
// status code : 0 : ok, -1 : non ok
// Type abstrait : list inserer(list l, int k, element e);
FONCTION insert(*pl : list, place :list , k : entier) : booléen
VAR noeudk, last, prec : list
DEBUT
  // la place concernée est-elle bien dans la liste ?
  SI ¬ appartient(place, *pl) ALORS
   RETOURNER faux
  FINSI
  noeudk ← newSingletionList(k) // création d'un nœud (cf p 87)
  // cas de la liste vide
  SI *pl= NULL ALORS
    *pl ← noeudk
    RETOURNER vrai
  FINSI
  // cas de la place en tête de liste
  SI place = *pl ALORS // noeudk mis en tête de liste
    noeudk→succ ← *pl
    noeudk→prec ← NULL
    *pl ← noeudk
   RETOURNER vrai
  FINSI
  // cas de la place en fin de liste
  SI place = NULL ALORS
    last ← lastElement(*pl) // on trouve le dernier élément de la liste
    last→succ ← noeudk
    noeudk→prev ← last
    RETOURNER vrai
  FINSI
  // autre cas : place en milieu de liste
  prec ← place→prev
  prec→succ ← noeudk
  noeudk→succ ← place
  nœud→kprev ← prec
  place→prev ← noeudk
  RETOURNER vrai
FIN
```

Implantation C

```
// Insertion dans une liste doublement chaînée
// pl : liste où s'effectue l'insertion
// place : place dans la liste (on insère juste devant)
```

```
// k : élément à insérer
// status code : 0 : ok, -1 : ko
// Type abstrait : list inserer(list l, int k, element e);
int insert(list* pl, list place, int k)
{
  // vérifier que la place est bien dans la liste concernée
  // (même méthode que pour l'exercice 2.5)
  if (!(areConvergent(place, *pl))) return -1;
  list K = newSingleton(k);
  // cas de la liste vide
  if (*pl == NULL)
  {
    *pl = K;
    return 0;
  }
  // cas tête de liste :
  if (place == *pl)
  {
    K->succ = *pl;
    K->prev = NULL;
    *pl = K;
    return 0;
  }
  // cas fin de liste :
  if (place == NULL)
  {
    list last = lastElement(*pl);
    last->succ = K;
    K->prev = last;
    return 0;
  }
  // sinon, milieu de liste
  list prec = place->prev;
  prec->succ = K;
  K->succ = place;
  K->prev = prec;
  place->prev = K;
  return 0;
}
```

- *Suppression*

Spécification de l'algorithme

```
// Suppression dans une liste doublement chaînée
// Type abstrait : list supprimer(list l, int k);
// status code : 0 : ok, -1 : ko
FONCTION removeElement(*pl :list, place : list) : booléen
VAR prec : list
DEBUT
  SI ¬ appartient(place, *pl) ALORS
   RETOURNER faux
  FINSI
  // cas de la liste vide
  SI isempty(*pl) ALORS // autre moyen de tester la liste vide
    RETOURNER vrai
  FINSI
  // cas de la place en tête de liste
  SI place = *pl ALORS
    *pl ← (*pl)→succ
    SI place→succ ≠ NULL ALORS
      (*pl)→succ→prev ← NULL
    FINSI
    LIBERER(place)
    RETOURNER vrai
  FINSI
  prec ← place→prev
  prec→succ ← place→succ
  SI place→succ ≠ NULL ALORS
    place→succ→prev ← prec
  FINSI
  LIBERER(place)
  RETOURNER vrai
FIN
```

Implantation C

```
// Suppression dans une liste doublement chaînée
// Type abstrait : list supprimer(list l, int k);
// status code : 0 : ok, -1 : ko
int removeElement(list* pl, list place)
{
  // vérifier que la place est bien dans la liste concernée
  // (même méthode que pour l'exercice 2.5)
  if (!(areConvergent(place, *pl))) return -1;
  // cas de la liste vide
```

```
    if (*pl == NULL) return 0;
    // cas tête de liste :
    if (place == *pl)
    {
      *pl = (*pl)->succ;
      if (place->succ != NULL) (*pl)->succ->prev = NULL;
      free(place);
      return 0;
    }
    list prec = place->prev;
    prec->succ = place->succ;
    if (place->succ != NULL) place->succ->prev = prec;
    free(place);
    return 0;
}
```

Corrigés des problèmes

Problème 3.1 Problème de Joseph

Étude

- *Phase de construction*

Avec $n = 12$ et $k = 3$:

- création de la liste circulaire ;
- création d'une liste chaînée avec les 12 éléments en croissante arithmétique de raison un en partant de un ; puis repli de cette liste sur elle-même grâce au `concat(&l, &l)`.

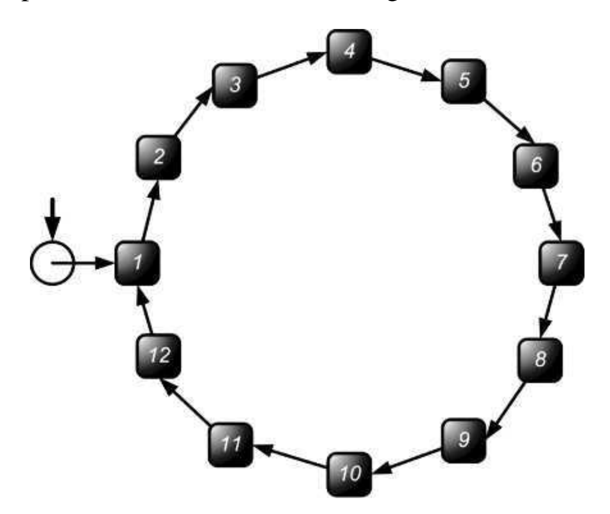


Figure 3.4 **Liste Joseph**

- *Phase de suppressions*

Suppressions jusqu'à épuisement de la liste : un nouvel ordre émerge, celui des éliminations. Appelons donc cette nouvelle liste Joséphine.

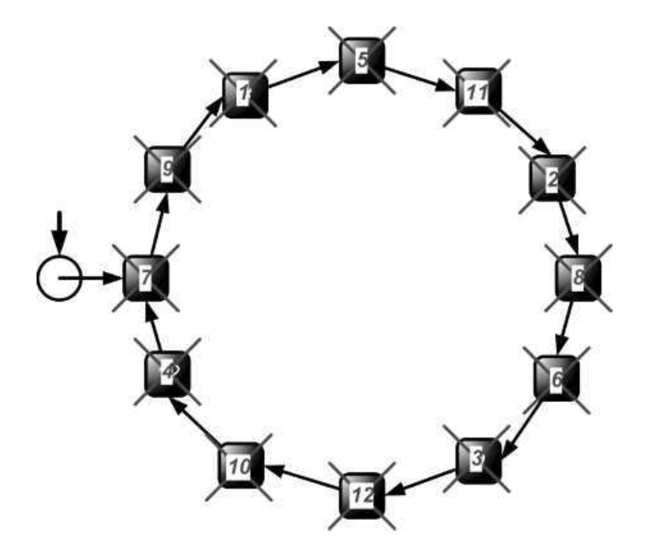


Figure 3.5 Liste Joséphine

- *Finalisation*

Pour la finalisation d'une liste circulaire doublement chaînée (reste deux, puis un élément, il convient d'être prudent) :

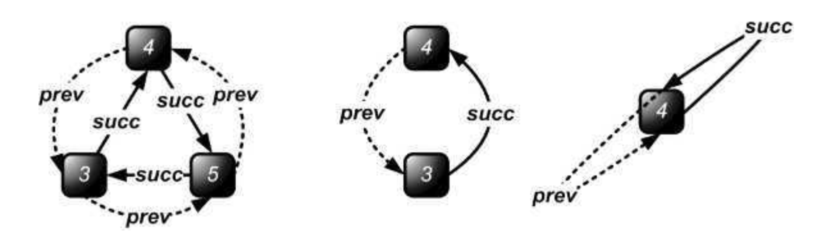


Figure 3.6 Finalisation d'une LDCC (Liste doublement chaînée circulaire)

Architecture fonctionnelle

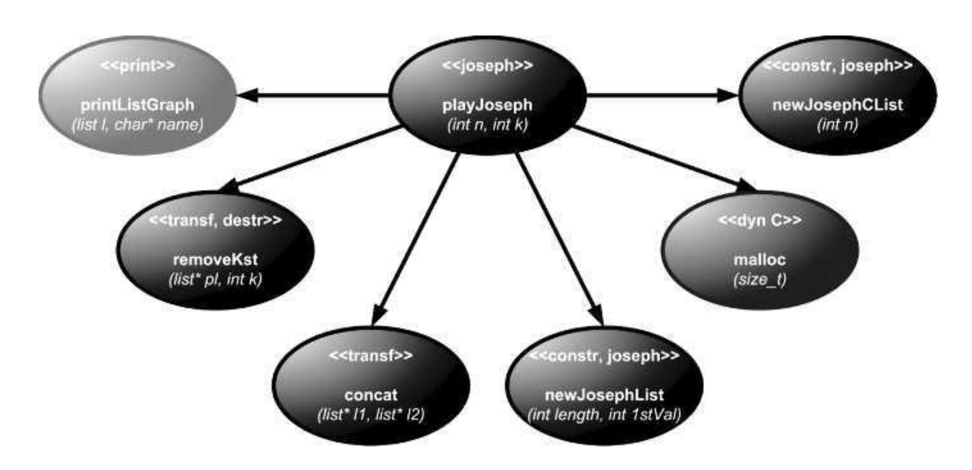


Figure 3.7 **Architecture fonctionnelle de PlayJoseph**

- *Fonction principale*

Spécification de l'algorithme

```
FONCTION playJoseph(n : entier, k : entier)
VAR joseph : list, i : entier
DEBUT
  AFFICHER("Joseph pour n=", n, " et k=", k)
  joseph ← newJosephCList(n)
  printListGraph(joseph, "1")
  i ← 2
  TANTQUE (joseph ≠ NULL) FAIRE
    joseph ← removeKst(&joseph, k)
    AFFICHER(i)
    i ← i+1
    printListGraph(joseph, "")
  FAIT
FIN
```

Implantation C

```
// Exécution du procédé d'élimination de Joseph
void playJoseph(int n, int k)
{
  printf("\nPlay Joseph for n=%d and k=%d\n", n, k);
  list joseph = newJosephCList(n);
  printListGraph(joseph, "1");
```

```
  int i = 2;
  while (joseph != NULL)
  {
    joseph = removeKst(&joseph, k);
    printf("%d", i++); printListGraph(joseph, "");
  }
}
```

• *Construction de la liste circulaire*

Spécification de l'algorithme

```
FONCTION newJosephCList(longueur : entier) : list
VAR joseph : list
DEBUT
  SI (longueur < 1) ALORS
    RETOURNER NULL
  FINSI
  joseph ← newJosephList(longueur, 1) // appel de la fonction qui suit
  concat(&joseph, &joseph)
  RETOURNER joseph
FIN
FONCTION newJosephList(longueur : entier, premier : entier) : list
VAR tête : list
DEBUT
  SI (longueur < 1) ALORS
    RETOURNER NULL
  FINSI
  RESERVER(tête)
  tête→contenu ← premier
  tête→succ ← newJosephList(longueur - 1, premier + 1)
  RETOURNER tête
FIN
```

Implantation C

```
// Création d'une liste circulaire de 1 à n (version récursive)
list newJosephCList(int longueur)
{
  if (longueur < 1) return NULL;
  list joseph = newJosephList(longueur, 1);
  concat(&joseph, &joseph);          // voir exercice 2.2
  return joseph;
}
```

```
list newJosephList(int longueur, int first)
{
  if (longueur < 1) return NULL;     // cas de la liste vide
  // initialisation de la chaîne avec la création de la tête
  list head = malloc(sizeof(place)); // allocation mémoire
  head->contenu = first;             // affectation du contenu
  head->succ = newJosephList(longueur - 1, first + 1);
  return head;                       // retour du pointeur sur la tête
}
```

- *Retrait du k^{ième} élément*

Spécification de l'algorithme

```
FONCTION removeKst(*pl : list, k :entier) : list
VAR tête, courant, prev : list
VAR circulaire : booléen ; périmètre : entier
DEBUT
  tête ⟵ *pl
  courant ⟵ *pl
  SI (courant = NULL) ALORS
    RETOURNER NULL
  FINSI
  circulaire ⟵ isCircular(courant)
  périmètre ⟵ getCLength(courant)
  SI (circulaire = vrai) ET (périmètre = 1) ALORS
    *pl ⟵ NULL
    RETOURNER *pl
  FINSI
  SI (circulaire = vrai) ALORS
    prev ⟵ getKstElement(*pl, périmètre-1)
  SINON
    prev ⟵ NULL
  FINSI
  TANTQUE (courant⟶succ ≠ NULL) ET (k > 1) FAIRE
    k ⟵ k-1
    prev ⟵ courant
    courant ⟵ courant⟶succ
  FAIT
  SI (courant⟶succ = NULL) ET (k > 0) ALORS
    RETOURNER *pl
  FINSI
  SI (circulaire = vrai) ALORS
    prev⟶succ ⟵ courant⟶succ
```

```
    SI (courant = tête) ALORS
      *pl ← courant→succ
    FINSI
    LIBERER(courant)
    RETOURNER prev→succ
  SINON
    SI (prev = NULL) ALORS
      *pl ← courant→succ
      LIBERER(courant)
      RETOURNER *pl
    SINON
      SI (courant→succ = NULL) ALORS
        prev→succ ← NULL
        LIBERER(courant)
        RETOURNER NULL
      SINON
        prev→succ ← courant→succ
        LIBERER(courant)
        RETOURNER prev→succ
      FINSI
    FINSI
  FINSI
FIN
```

Implantation C

```
// Supprime le kième et retourne le suivant
//   Si le 1er élément est supprimé retour du suivant sinon *pl inchangé
//   note : la fonction fonctionne pour les listes
//          circulaires et linéaires simplement chaînées
list removeKst(list* pl, int k)
{
  list head = *pl;
  list curr = *pl;
  if (curr == NULL) return NULL;
  // pour moduler les traitements : liste circulaire ou linéaire ?
  int circular = isCircular(curr);
  int perimeter = getCLength(curr);
  if (circular & perimeter == 1) // finalisation
  {
    *pl = NULL;
    return NULL;
  }
  // mémorisation du prev : soit le prev de curr, soit NULL (linéaire)
```

```
list prev = (circular?getKstElement(*pl, perimeter - 1):NULL);
// on commence par itérer jusqu'à la bonne position.
// note : & != && : le k est décrémenté ssi curr->succ != NULL
while (curr->succ != NULL & --k > 0)
{
  prev = curr;
  curr = curr->succ;
}
// si la liste n'est pas circulaire
// et qu'on a atteint la queue alors que k > 0, on arrête
if (curr->succ == NULL && k > 0) return *pl;
// sinon : on supprime en tenant compte de tous les cas possibles
// cas de la liste circulaire
if (circular)
{
  prev->succ = curr->succ;
  // danger : si on retire le 1er : *pl pointe alors dans le vide
  if (curr == head) *pl = curr->succ;
  free(curr);
  return prev->succ;
}
// liste linéaire ---* la suite n'est pas indispensable pour Joseph
else
{
  // tete d'une linéaire
  if (prev == NULL)
  {
    *pl = curr->succ;
    free(curr);
    return *pl;
  }
  else
  {
    if (curr->succ == NULL)  // queue d'une linéaire
    {
      prev->succ = NULL;
      free(curr);
      return NULL;
    }
    else  // milieu d'une linéaire : comme pour une circulaire
    {
      prev->succ = curr->succ;
      free(curr);
```

```
        return prev->succ;
      }
    }
  }
}
```

Problème 3.2 Mesurer la longueur d'une liste de type Heqat

Spécification de l'algorithme

```
FONCTION getHLength(l :list l, *longueurLinéaire : entier,
*longueurCirculaire : entier): entier
VAR longLigne, longueur : entier ; tête, t2 : list
DEBUT
  SI (l = NULL) ALORS
    *longueurLinéaire ← 0
    *longueurCirculaire ← 0
    RETOURNER 0
  FINSI
  SI (l→succ = l) ALORS
    *longueurLinéaire ← 0
    *longueurCirculaire ← 1
    RETOURNER 1
  FINSI
  longLigne ← 0
  longueur ← 1
  tête ← l
  TANTQUE (l→succ ≠ NULL) ET (l→succ ≠ tête) FAIRE
    l ← l→succ
    longueur ← longueur+1
    SI (l→succ ≠ NULL) ET (l→succ = tête) ALORS
      *longueurLinéaire ← 0
      *longueurCirculaire ←longueur
      RETOURNER longueur
    FINSI
    t2 ← tête
    longLigne ← 0
    TANTQUE (t2 ≠ l) ET (t2→succ ≠ NULL) FAIRE
      t2 ← t2→succ
      longLigne ← longLigne+1
      SI (l→succ ≠ NULL) ET (l→succ = t2) ALORS
        *longueurLinéaire ← longLigne
        *longueurCirculaire ← longueur-longLigne
        RETOURNER longueur
      FINSI
```

```
    FAIT
  FAIT
  *longueurLinéaire ← longueur
  *longueurCirculaire ← 0
  RETOURNER longueur
FIN
```

Implantation C

```
/**
 * Retourne la longueur totale de la liste, et précise,
 * en modifiant les paramètres lineLength et loopLength,
 * les longeurs respectives du segment amont (éventuel)
 * et de la partie circulaire aval (éventuelle)
 * Les valeurs possibles sont :
 * 0 = 0 + 0 : liste vide
 * l = l + 0 : liste linéaire (longueur)
 * p = 0 + p : liste circulaire (périmètre)
 * n = l + p : liste heqat, composée d'une liste linéaire en amont
 *   et d'une liste circulaire en aval
 */
int getHLength(list l, int* lineLength, int* loopLength)
{
  /// cas de la liste vide
  if (l == NULL)
  {
    *lineLength = 0;
    *loopLength = 0;
    return 0;
  }
  /// à partir de là, il est certain qu'il existe au moins un élément
  /// cas du sigleton circulaire
  if (l->succ == l)
  {
    *lineLength = 0;
    *loopLength = 1;
    return 1;
  }
  /// c'est à partir de là que le vrai travail commence
  int lineL = 0;  /// la partie linéaire préfixe mesure au moins 'un'
  int length = 1;
  /// on mémorise la tête
  list head = l;
  /// tant qu'il est possible d'itérer sans retomber sur la tête :-)
```

```
    while (l->succ != NULL && l->succ != head)
    {
      /// iteration : 1er passage, l pointe sur le second élément
      l = l->succ;
      length++;
      /// s'il existe au moins un suivant,
      /// et si alors, ce troisième élément est la tête
      /// c'est qu'on a une circulaire (en l'occurrence, le triangle)
      if (l->succ != NULL) if (l->succ == head)
      {
        *lineLength = 0;
        *loopLength = length;
        return length;
      }
      /// sinon on remet le pointeur h à zéro,
      /// i.e., on le refait pointer sur la tête
      list h = head;
      lineL = 0; /// on reprend le décompte
      /// tant que h n'a pas rattrapé l et qu'il reste itérable..
      while ((h != l) && h->succ != NULL)
      {
        h = h->succ; // .. itérons le donc
        lineL++;
        /// si l est itérable et si alors son suivant EST h :
        /// il y a boucle !!
        if (l->succ != NULL) if (l->succ == h)
        {
          *lineLength = lineL;
          *loopLength = length - lineL;
          return length;
        }
      }
    }
    /// si on arrive là, c'est une linéaire classique terminant en NULL
    *lineLength = length;
    *loopLength = 0;
    return length;
}
```

STRUCTURES ARBORESCENTES

4

RAPPELS DE COURS

Un **arbre** est un ensemble de **nœuds**, organisés de façon hiérarchique, à partir d'un nœud distingué, appelé **racine**. La structure d'arbre est l'une des plus importantes et des plus spécifiques de l'informatique : par exemple, c'est sous forme d'arbre que sont organisés les fichiers dans des systèmes d'exploitation tels qu'UNIX ; c'est aussi sous forme d'arbres que sont représentés les programmes traités par un compilateur...

Une propriété intrinsèque de la structure d'arbre est la récursivité, et les définitions des caractéristiques des arbres, aussi bien que les algorithmes qui manipulent des arbres s'écrivent très naturellement de manière récursive.

4.1 ARBRES BINAIRES

Examinons tout d'abord quelques exemples simples représentés par des arbres binaires :

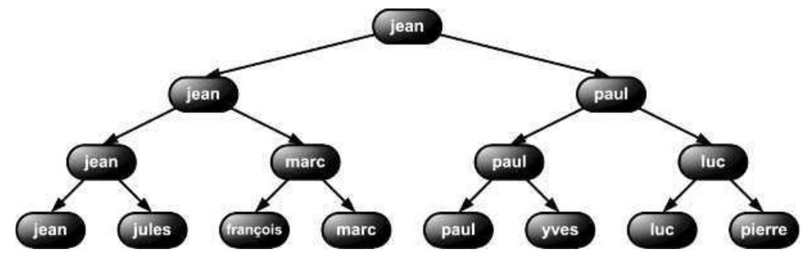


Figure 4.1
Les résultats d'un tournoi de tennis : au premier tour Jean a battu Jules, Marc a battu François, Paul a battu Yves, et Luc a battu Pierre ; au deuxième tour Jean a battu Marc, et Paul a battu Luc ; et Jean a gagné en finale contre Paul.

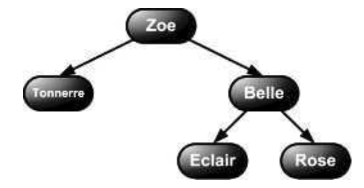


Figure 4.2
Le pedigree d'un cheval Zoe ; son père est Tonnerre et sa mère est Belle ; la mère de Belle est Rose et le père de Belle est Éclair...

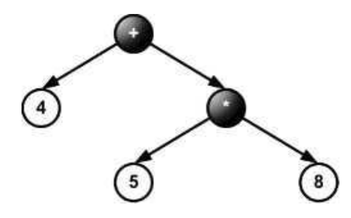


Figure 4.3
Une expression arithmétique dans laquelle tous les opérateurs sont binaires...
4 + 5 * 8

La structure d'arbre binaire est utilisée dans très nombreuses applications informatiques ; de plus les arbres binaires permettent de représenter les arbres plus généraux.

4.1.1 Définition

Le vocabulaire concernant les arbres informatiques est souvent emprunté à la botanique ou à la généalogie ; étant donné un arbre **B = <o, B1, B2>** :

- 'o' est la **racine** de B.
- B1 est le **sous-arbre gauche** (sag) de la racine de B, (ou, plus simplement, le sous-arbre gauche de B), et B2 est son **sous-arbre droit** (sad).
- On dit que C est un **sous-arbre** de B si, et seulement si : C = B, ou C = B1, ou C = B2, ou C est un sous-arbre de B1 ou de B2.
- On appelle **fils gauche** (respectivement **fils droit**) d'un nœud la racine de son sous-arbre gauche (respectivement sous-arbre droit), et l'on dit qu'il y a un **lien gauche** (respectivement droit) entre la racine et son fils gauche (respectivement fils droit).
- Si un nœud n_i a pour fils gauche (respectivement droit) un nœud n_j, on dit que n_i est le **père** de n_j (chaque nœud n'a qu'un seul père).
- Deux nœuds qui ont le même père sont dits **frères**.
- Le nœud n_i est un **ascendant** ou un **ancêtre** du nœud n_j si, et seulement si, n_i est le père de n_j, ou un ascendant du père de n_j; n_i est un **descendant** de n_j si, et seulement si n_i est fils de n_j, ou n_i est un descendant d'un fils de n_j.

Tous les nœuds d'un arbre binaire ont au plus deux fils :

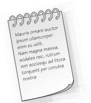
- un nœud qui a deux fils est appelé **nœud interne** ou **point double**
- un nœud sans fils est appelé **nœud externe** ou **feuille**.

Donc, un arbre binaire est :

- soit vide ;
- soit constitué d'un élément de type T et d'au plus deux arbres binaires.

4.1.2 Représentation

La représentation la plus naturelle reproduit la définition récursive des arbres binaires. Elle peut être réalisée en allouant la mémoire soit de façon chaînée soit de façon contiguë.



La représentation classique d'un arbre est dynamique.

À chaque nœud on associe deux pointeurs, l'un vers le sous-arbre gauche, l'autre vers le sous-arbre droit, et l'arbre est déterminé par l'adresse de sa racine. Lorsque l'arbre est étiqueté, on représente dans un champ supplémentaire l'information contenue dans le nœud.

Implantation en C

```
typedef struct arbre {
  T info;
  struct arbre *sag, *sad;
} arbre;
typedef arbre *ptr_arbre;
```

Si a est un pointeur sur la racine de l'arbre, alors `a = NULL` correspond à un arbre vide ; `a->sag` pointe sur le fils gauche de `a` ; `a->sad` pointe sur le fils droit de `a` ; `a->info` permet d'accéder au contenu du nœud.

On ne parle d'arbres binaires que par l'intermédiaire des algorithmes qui utilisent le type arbre.

4.1.3 Algorithmes de parcours d'un arbre binaire

Il y a deux types de parcours :

- **Parcours en profondeur d'abord –** Examiner complètement un chemin et passer au chemin suivant tant qu'il en reste.

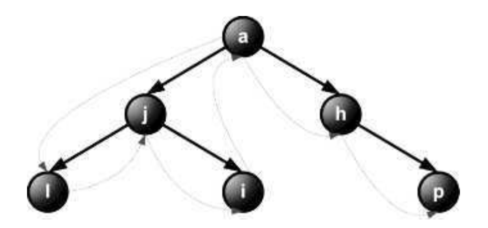


Figure 4.4 **Parcours en profondeur (à l'aide d'une pile)**

- **Parcours en largeur d'abord** – Examiner tout un niveau (profondeur hiérarchique) passant au niveau du dessous tant qu'il en reste.



Problème : pas de lien entre fils. Cela doit être traité itérativement.

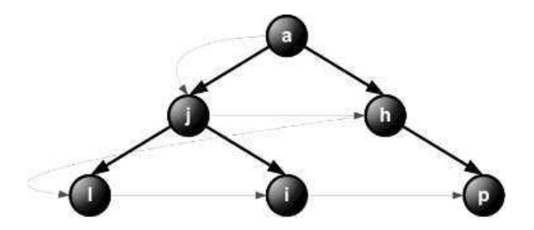


Figure 4.5 Parcours en largeur (à l'aide d'une file)

Parcours en profondeur d'abord

Pré-ordre

Principe

```
Si arbre non vide alors :
  traiter la racine
    parcourir en Pré-ordre le sag
    parcourir en Pré-ordre le sad
```

Algorithme en pseudo C
Donnée : ptr_arbre p

```
FONCTION préordre(a : ptr_arbre)
DEBUT
  SI a ≠ NULL ALORS
    afficher(a→info)
    préordre(a→sag)
    préordre(a→sad)
  FINSI
FIN
```

Exemple

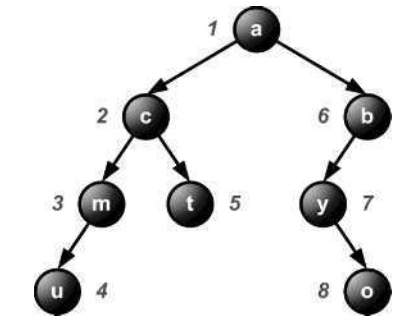
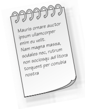


Figure 4.6 Parcours en préordre

On affiche : a c m u t b y o



Il n'y a pas de priorité sur le parcours des sous-arbres. On pourrait commencer par traiter le sad avant le sag.

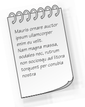
Ordre

Principe

```
SI arbre est non vide alors :
  parcourir le sag
  traiter la racine
  parcourir le sad
```

Algorithme
Donnée : ptr_arbre a

```
FONCTION ordre(a : ptr_arbre)
DEBUT
  SI a ≠ NULL ALORS
    ordre(a→sag)
    afficher(a→info)
    ordre(a→sad)
  FINSI
FIN
```



Cet algorithme nous permet d'obtenir les informations dans un ordre total. On peut rentrer les informations dans un arbre binaire de façon triée.

Exemple

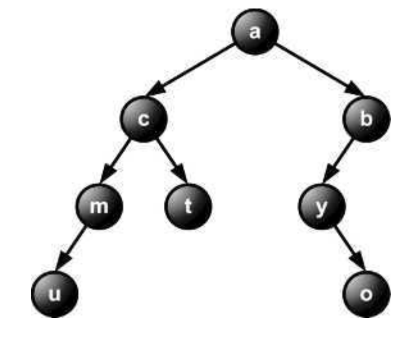


Figure 4.7 **Parcours en ordre**

On affiche : u m c t a y o b

Postordre

```
SI arbre n'est pas vide :
  parcourir en post-ordre le sag
  parcourir en post-ordre le sad
  traiter la racine
```

Algorithme
Donnée : ptr_arbre a

```
FONCTION postordre(a : ptr_arbre)
DEBUT
  SI a ≠ NULL ALORS
    postordre(a→sag)
    postordre(a→sad)
    afficher(a→info)
  FINSI
FIN
```

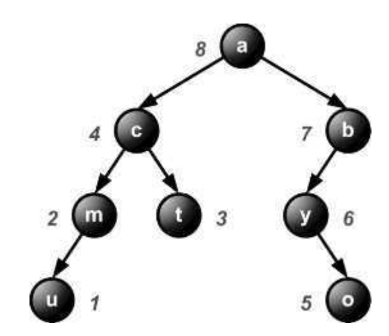


Figure 4.8 **Parcours en postordre**

On affiche : u m t c o y b a

4.1.4 Arbres binaires de recherche (ABOH = Arbres Binaires Ordonnés Horizontalement)

Définitions et algorithmes de manipulation

ABOH = Arbre Binaire Ordonné Horizontalement – Il est tel que pour tout nœud de l'arbre, les éléments de son sag lui sont inférieurs, de son sad lui sont supérieurs ou égaux.

Exemple

On a bien l'arbre ordonné horizontalement.
Pour un parcours en ordre on obtient 1 5 6 10 15 30 35 50 60
La relation d'ordre est donc totale.

Mais cela dépend de la définition, on peut mettre les plus grands dans le sad ou sag.

Ordre :
croissant : sag < racine <= sad
décroissant : sad >= racine > sag

Figure 4.9 **Arbre Binaire Ordonné Horizontalement (ABOH)**

Algorithmes de manipulation de l'ABOH

Ordre – Donne les éléments en ordre total croissant ou décroissant selon la définition choisie pour ABOH et la priorité de traitement du sag par rapport au sad.

• *Recherche d'un élément dans un ABOH*

Pas de notion de retour arrière, on ne parcourt qu'une branche de l'arbre car on sait s'il est plus grand ou plus petit. Donc, le parcours est naturellement dichotomique.

Principe

```
SI l'arbre est vide ALORS
  fin et échec
SINON
  SI l'élément cherché = élément pointé ALORS
    fin et réussite
SINON
  SI l'élément cherché < élément pointé ALORS
    rechercher l'élément dans le sag
  SINON
    rechercher l'élément dans le sad
```

Algorithme
Donnée : ptr_arbre a, T x
Résultat de type booléen

```
FONCTION recherche(x : T, a : ptr_arbre) : booléen
VAR ok : booléen
DEBUT
  SI a = NULL ALORS
    ok ← faux
  SINON
    SI a→info = x ALORS
      ok ← vrai
    FINSI
  SINON
    SI a→info > x ALORS
      recherche(x, a→sag, ok)
    SINON
      recherche(x, a→sad, ok)
    FINSI
  FINSI
  RETOURNER ok
FIN
```

Cet algorithme renvoie la première occurrence du terme cherché, c'est-à-dire qu'il ne considère que la première fois où il rencontre l'élément cherché.

Si on recherche la $n^{\text{ième}}$ occurrence de l'élément dans l'arbre, il faut utiliser un compteur.

• *Ajout d'un élément dans un ABOH*

Principe

```
Placer l'élément dans l'arbre en conservant l'ordre
et faisant le moins de réorganisation possible.
Un ajout d'élément dans un ABOH se fait systématiquement aux feuilles :
  SI arbre est vide ALORS création et ajout
  SI non vide ALORS trouver la feuille
```

Trouver la feuille : parcourir l'arbre et rechercher la position de l'élément c'est-à-dire comparer l'élément à ajouter à la racine.

```
SI racine > élément à ajouter ALORS
  ajout dans le sag, donc retour en 1) avec le sag.
SI racine ⩽ élément à ajouter ALORS
  ajout dans le sad, donc retour en 1) avec le sad
```

Exemple

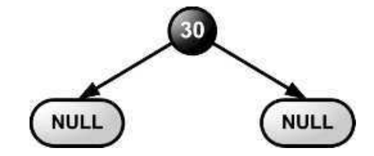


Figure 4.10

On veut ajouter 10 et 50

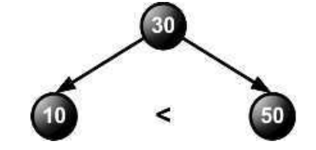


Figure 4.11

On veut ajouter 5 et 15

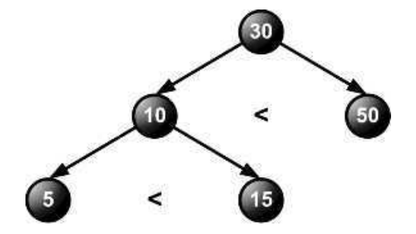


Figure 4.12

On veut ajouter 6 et 12

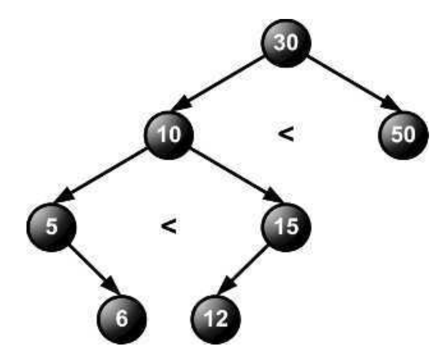
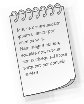


Figure 4.13



– Recherche dichotomique de la position.
– Aucune réorganisation à produire – on se contente d'ajouter un élément.
– En statique, ajouter un élément, c'est ajouter un élément dans le tableau.

Algorithme (en récursif)
Donnée : T x
Donnée modifiée : ptr_arbre *aa

```
FONCTION ajout(x : T, *aa : arbre)
DEBUT
  SI *aa = NULL ALORS
    reserver(*aa)
    *aa→info ← x
    *aa→sag ← NULL
    *aa→sad ← NULL
  SINON
    SI *aa→info ⩽ x ALORS
      ajout(x, &(*aa→sad))
    SINON
      ajout(x, &(*aa→sag))
    FINSI
  FINSI
FIN
```

*aa est en donnée modifiée, donc on gère bien le lien avec la récursivité.



1) Le mode de transmission par référence crée automatiquement le lien entre le père et le fils
2) Ce qui ne marche pas :

```
q ← *aa→sad
ajout(x, &q)
```

Dans ce cas le lien est cassé, car q est une variable locale. **C'est ce qu'il ne faut surtout pas faire !!!**

- *Suppression d'un élément dans un ABOH*

L'élément est une feuille

```
Suppression simple :
  libération mémoire
  mise à jour du pointeur concerné dans le père
```

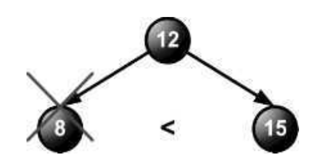


Figure 4.14

L'élément est le père d'un seul sous-arbre

```
Mise à jour du pointeur concerné dans le père de l'élément supprimé
avec l'adresse du sous-arbre de l'élément supprimé.
```

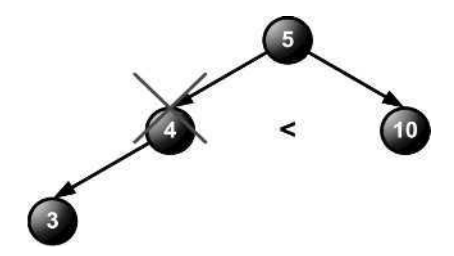


Figure 4.15

L'élément à deux sous-arbres

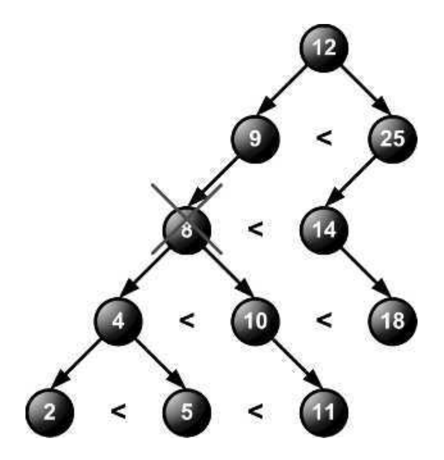


Figure 4.16 **On veut supprimer le 8**

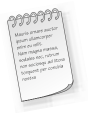
```
Rechercher le plus grand du sag (ou le plus petit du sad)
Recopier sa valeur à la place de l'élément à supprimer
Supprimer le plus grand du sag (ou le plus petit du sad)
par la méthode 1 ou 2 (feuille ou un seul sous-arbre)
```

Principe

```
Si l'arbre est vide, retourner faux
Si l'information est plus petite à l'élément, retour en 1) avec le sad
Si l'information est plus grande à l'élément, retour en 1) avec le sag
```

```
Si info = élément (on a trouvé l'élément à supprimer)
  Ni sag, ni sad → libération et retour avec vrai
  sad et non sag ou sag et non sad → mise à jour du père,
  libération et retour avec vrai
  sag et sad →recherche du plus petit élément dans le sad
    →remplacement d'élément par le plus petit
    →suppression du plus petit avec a) ou b)
    →retour avec vrai
```

Algorithmes sur l'équilibre des arbres binaires



Déséquilibre possible d'un ABOH.

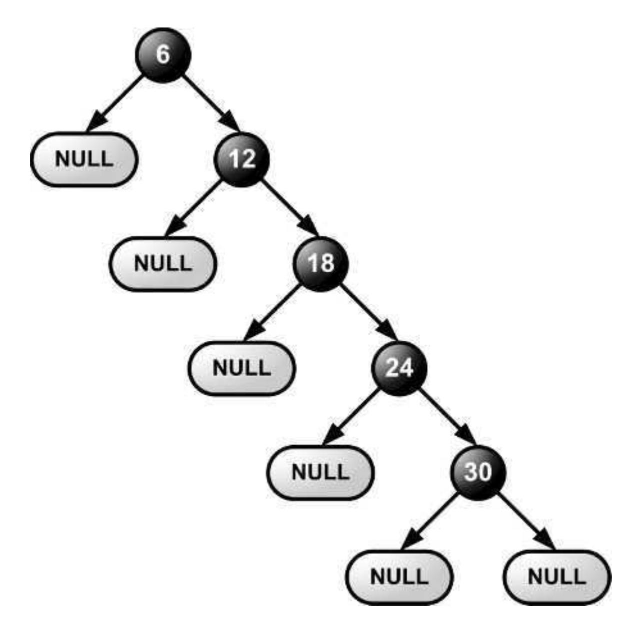


Figure 4.17 **Arbre totalement déséquilibré**

Si un arbre est déséquilibré, ses performances sont instables. Les performances dépendent de la façon d'entrer les informations.

La recherche n'est plus réellement dichotomique dans un arbre déséquilibré.

- *Définition d'un arbre équilibré*

Équilibre parfait : pour tout nœud de l'arbre, la valeur absolue de la différence entre le nombre des nœuds du sad et le nombre des nœuds du sag est inférieure ou égale à 1.

$$|n_g - n_d| \leqslant 1$$

Exemples

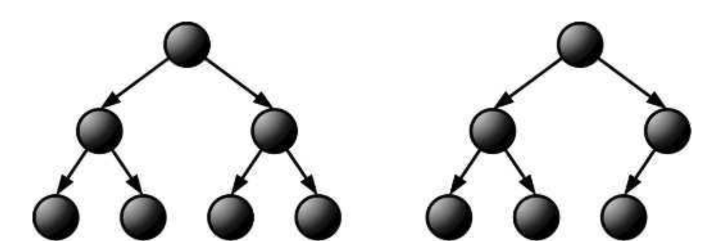


Figure 4.18 **Arbres parfaitement équilibrés**

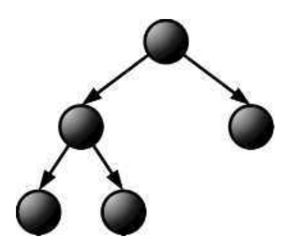
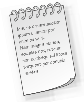


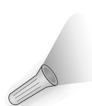
Figure 4.19 **Arbre imparfaitement équilibré**

Équilibre partiel : pour tout nœud de l'arbre, la valeur absolue de la différence entre la hauteur du sad et la hauteur du sag est inférieure ou égale à 1.

$$|h_g - h_d| \leqslant 1$$



Les trois arbres de l'exemple précédent sont partiellement équilibrés.



Il existe des procédés pour éviter le déséquilibre d'arbres : par exemple, les arbres AVL.

Principe

```
SI l'arbre est vide → retourner 0.
SINON compter le nombre de nœuds du sag
compter le nombre de nœuds du sad
retourner 1 + nsag + nsad
```

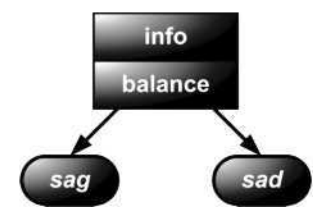


Figure 4.20
Balance :
-1 : hsag = hsad + 1
0 : hsad=hsag
+1 : hasd = hsag + 1

Algorithme
Donnée ptr_arbre a
Résultat de type entier

```
FONCTION compter(a : ptr_arbre) : entier
VAR  n :entier
DEBUT
  SI a = NULL ALORS
    n ← 0
  SINON
    n ← 1 + compter(a→sag) + compter(a→sad)
  FINSI
  RETOURNER n
FIN
```

Hauteur

```
SI l'arbre est vide → retourner 0.
SINON calculer la hauteur du sag
calculer la hauteur du sad
SI hg > hd alors retourner 1 + hg
SINON retourner 1 + hd
```

Algorithme
Donnée ptr_arbre a
Résultat de type entier

```
FONCTION hauteur(a : ptr_arbre) : entier
VAR  n :entier
DEBUT
  SI a = NULL ALORS
    n ← 0
```

```
  SINON
    n ⟵ 1 + maximum(compter(a⟶sag), compter(a⟶sad))
  FINSI
  RETOURNER n
FIN
```

Équilibre parfait

```
-  SI l'arbre est vide, il est parfaitement équilibré
-  SINON  compter le nombre de nœuds du sag
compter le nombre de nœuds du sad
SI |ng -- nd| > 1 → retourner faux
SINON vérifier l'équilibre parfait du sag
SI oui → vérifier l'équilibre parfait du sad
  SI oui → retourner vrai
  SINON → retourner faux
SINON → retourner faux
```

Algorithme
Donnée ptr_arbre a
Résultat de type entier

```
FONCTION équilibre_parfait(a : ptr_arbre) : booléen
VAR  ok : booléen ; cptgauche, cptdroit : entier
DEBUT
  SI a = NULL ALORS
    ok ⟵ vrai
  SINON
    cptgauche ⟵ compter(a⟶sag)
    cptdroit ⟵ compter(a⟶sad)
    SI |cptgauche -- cptdroit| > 1 ALORS
      ok ⟵ faux
    SINON
      SI équilibre_parfait(a⟶sag) ET équilibre_parfait(a⟶sad) ALORS
        ok ⟵ vrai
      SINON
        ok ⟵ faux
      FINSI
    FINSI
  FINSI
  RETOURNER ok
FIN
```

ÉNONCÉS DES EXERCICES ET DES PROBLÈMES

EXERCICES

Exercice 4.1 Traiter un arbre en post-ordre *

- a) Écrire un algorithme récursif de traitement en post-ordre d'un arbre binaire.
- b) Dérouler l'algorithme sur l'exemple suivant :

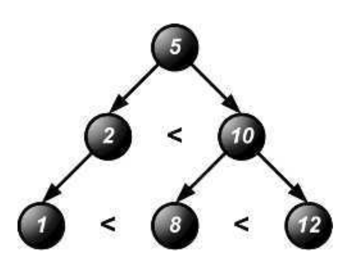


Figure 4.21

Exercice 4.2 Afficher le contenu des feuilles d'un arbre **

Écrire un algorithme permettant d'afficher tous les entiers enregistrés dans les feuilles d'un arbre binaire (en ignorant les entiers contenus dans tous les autres nœuds).

Exercice 4.3 Rechercher itérativement un élément dans un ABOH *

Écrire un algorithme itératif de recherche d'un élément dans un ABOH.

Exercice 4.4 Évaluer le caractère ABOH d'un AB ***

Écrire un algorithme permettant de déterminer si un arbre binaire donné est ou n'est pas un arbre ABOH.

Exercice 4.5 Mesurer la hauteur d'un ABOH **

Écrire un algorithme de calcul de la hauteur d'un ABOH.

Exercice 4.6 Évaluer l'équilibre d'un AB ***

Écrire un algorithme permettant de savoir si un arbre binaire est partiellement équilibré.

Exercice 4.7 Parcourir un AB en largeur et en profondeur ***

Écrire deux algorithmes, respectivement de parcours en largeur d'abord et en profondeur d'abord, pour afficher le contenu d'un arbre binaire.

Exercice 4.8 Effectuer un calcul complexe sur un arbre ****

Écrire un algorithme qui retourne la moyenne des éléments positifs et celle des éléments négatifs d'un arbre (0 est considéré ici comme un positif).

Exercice 4.9 Extraire une liste d'éléments d'un arbre ****

Écrire un algorithme qui retourne la liste chaînée des éléments d'un arbre d'entiers, qui ne sont pas divisibles par leur parent et que leur parent ne divise pas.

Citez au moins deux façons de construire des arbres pour lesquels cet algorithme retourne nécessairement la liste de tous ses éléments.

Exercice 4.10 Produire des coupes transversales d'un arbre ***

Coupe transversale d'un arbre sur un niveau donné, restituée sous forme de liste.

Écrire un algorithme qui retourne la coupe transversale.

PROBLÈMES

Problème 4.1 Le triangle Chinois de Pascal ****

Isaac Newton a donné une formule générale du développement de la puissance $n^{\text{ième}}$ d'un binôme :

$$(x + y)^n = \sum_{k=0}^{n} C_n^k x^{n-k} y^k$$

où C_n^k, le « coefficient binomial », alternativement dénoté par $\binom{n}{k}$, se calcule par la formule :

$$C_n^k = \frac{n!}{k!\,(n-k)!} = C_n^{n-k}$$

En combinatoire, C_n^k décompte le nombre de combinaisons de k éléments parmi n.

Le triangle de Pascal, ou « triangle arithmétique Chinois » date du XIIIe siècle et n'a donc pas été inventé par notre Blaise national.

Vous avez certainement déjà rencontré cet objet mathématique qui exploite une méthode de construction par récurrence pour produire un arrangement géométrique arborescent des coefficients binomiaux.

En effet, il est trivial de constater que : $C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$ avec pour chaque ligne :

$$C_n^0 = \frac{n!}{0!n!} = C_n^n = 1$$

et en particulier pour la première :

$$C_0^0 = \frac{0!}{0!0!} = C_0^0 = 1$$

Il vous est demandé de réaliser l'algorithme itératif qui construit le triangle sous forme d'un arbre binaire non équilibré, jusqu'à un niveau de profondeur n donné, dont la racine contient la valeur C_0^0, le fils gauche de la racine C_1^0, le fils droit de la racine C_1^1, puis, dès la profondeur $n = 2$, dont les fils gauche et droit du fils le plus à gauche de profondeur n – 1 (contenant C_{n-1}^0) contiennent respectivement les valeursC_n^0 et C_n^1, et dont tous les nœuds situés à droite de ces deux fils (contenant $C_n^{k>1}$) soient les fils droits de leur père de niveau n – 1 (contenant $C_{n-1}^{k-1>0}$).

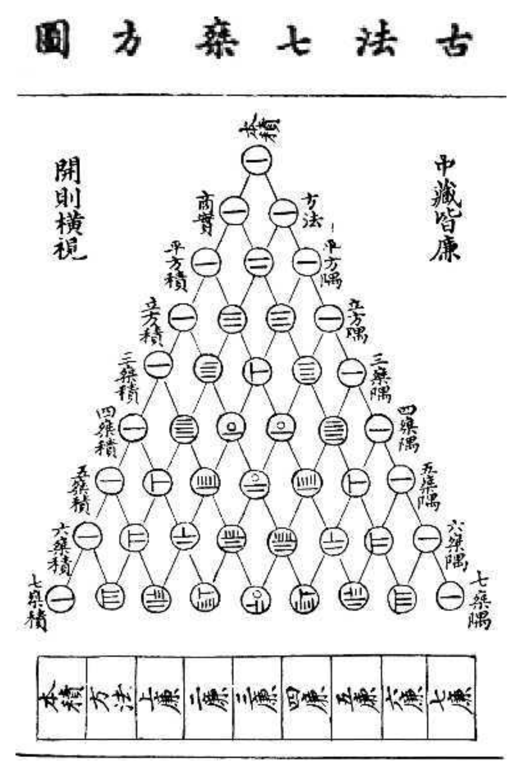


Figure 4.22 **Triangle de Yang Hui – Source : http://en.wikipedia.org/wiki/File:Yanghui_triangle.gif**

Le .h vous est donné, et l'algorithme peut être réalisé en pseudo code ou directement en C :

```
typedef struct treeNode
{
  int contenu;
  struct treeNode *sag, *sad;
} treeNode;
typedef treeNode *tree;
void buildPascalTriangle(tree* t, unsigned n);
```

Problème 4.2 Interlude Syracusien *****

Une suite de Syracuse de terme initial un entier N est définie par :

$$s_0^N = N$$

$$\forall n \in \mathbb{N} : \begin{cases} \text{pair}\left[s_n^N\right] \rightarrow s_{n+1} = \dfrac{s_n^N}{2} \\ \text{impair}\left[s_n^N\right] \rightarrow s_{n+1} = 3s_n^N + 1 \end{cases}$$

Il semblerait que quel que soit N, la suite s^N converge vers 1, i.e. :

$$\forall N \in \mathbb{N} : \exists q \in \mathbb{N}/s_q^N = 1$$

Dans cet interlude, il vous est demandé de réaliser une fonction récursive qui construit un arbre de Syracuse de racine un entier donné e, de profondeur donnée q, dont les branches sont toutes les suites de Syracuse qui convergent vers e en q étapes.

Pour cela, il suffit de procéder à une inversion de la suite.

Partons de e, le terme de convergence visé, et racine de l'arbre : e peut être le successeur direct de $2e$, et éventuellement de $(e - 1)/3$ si cet entier existe et s'il est impair. Disons que e a nécessairement un fils droit $2e$, et éventuellement un fils gauche $(e - 1)/3$. En appliquant récursivement ce procédé, on peut donc construire un arbre binaire de profondeur quelconque dont chaque branche est l'une des suites de Syracuse qui converge vers e.

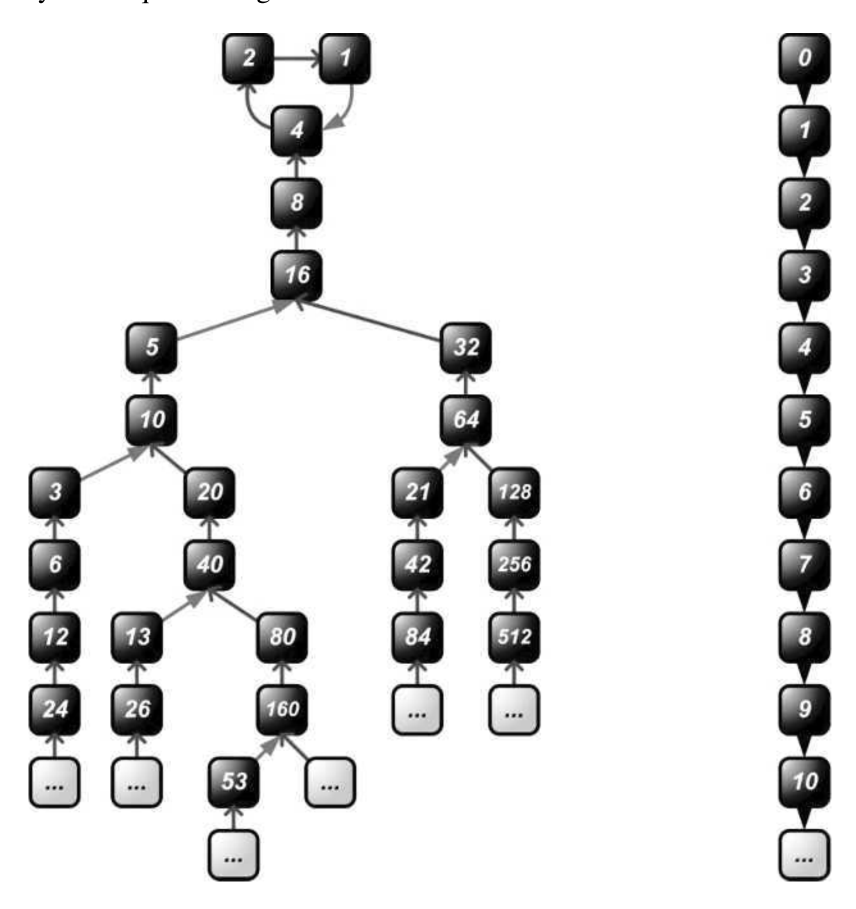


Figure 4.23 Arbre de Syracuse

Écrire une fonction qui retourne le prédécesseur entier impair d'un entier donné s'il existe, et 0 sinon.

Écrire une fonction récursive qui construit l'arbre de Syracuse à partir d'une feuille donnée contenant la valeur *e* et développé jusqu'à la profondeur *p*. Si un fils gauche (e – 1)/3 vaut 0 ou 1 à une étape quelconque de la récursion (i.e. *e* vaut 1 ou 4), ce fils gauche n'est pas créé.

Corrigés des exercices et des problèmes

EN PRÉAMBULE

Quelques remarques utiles

Les structures arborescentes, bien que plus complexes que les structures séquentielles, conduisent pardoxalement à des algorithmes naturellement récursifs et donc plus « compacts ».

Implantation en C d'une structure d'arbre binaire

Pour la réalisation en C de tous les algorithmes spécifiés ci-dessous, on définit la structure d'arbre binaire suivante dont on précisera au cas pas cas, le type <element> (prototype) :

Pseudo code formel

```
STRUCTURE treeNode
  content : <élément>
  *sag, *sad : treeNode
TYPE *tree : treeNode
TYPE *ptree : tree
```

Implantation en C

```
typedef struct treeNode
{
  <élément> content;
  struct treeNode *sag, *sad;
} treeNode;
typedef treeNode *tree;
typedef tree *ptree;
```

Qelques fonctions utilitaires facultatives pour améliorer la lisibilité algorithmique de vos implémentations :

```
tree newSingletonTree(unsigned content)
{
  tree t = malloc(sizeof(treeNode));  // allocation mémoire
  t->content = content;               // affectation du contenu
```

```
    t->sag = NULL;
    t->sad = NULL;
    return t;
  }
  int isEmptyTree(tree t) {return (t == NULL);}
  int isSingletonTree(tree t) {
    return isEmptyTree(t->sag) && isEmptyTree(t->sad);
  }
```

La structure de liste simplement chaînée du second chapitre, et en particulier la fonction `concat` de l'exercice 2.2 sont régulièrement réutilisées.

Corrigés des exercices

Exercice 4.1 traiter un arbre en post-ordre

Étude

Il s'agit d'une application directe des rappels de cours.

Pour rester générique, nous interprétons l'énoncé en retournant une liste chaînée des éléments de l'arbre ordonnés en post-ordre. Cette liste peut ensuite faire l'objet de tout traitement itératif comme par exemple pour en afficher le contenu.

Le résultat produit sur l'exemple donné est : 1, 2, 8, 12, 10, 5.

Spécification de l'algorithme

```
FONCTION getPostorderRouteList(t : tree) : list
VAR thisNode, sagldpl, sadldpl : list
DEBUT
  // préconditions et prétraitements standards des cas aux limites
  SI (isEmptyTree(t)) ALORS
    RETOURNER emptyList()
  FINSI
  thisNode ← newSingletonList(t→content)
  SI (isSingletonTree(t)) ALORS
    RETOURNER thisNode
  FINSI
  sagldpl ← getPostorderRouteList(t→sag)
  sadldpl ← getPostorderRouteList(t→sad)
  RETOURNER *concat(&sagldpl, concat(&sadldpl, &thisNode))
FIN
```

Exercice 4.2 afficher le contenu des feuilles d'un arbre

Étude

À défaut de prescription contraire, nous élaborons un algorithme récursif.

Le critère caractéristique d'un nœud feuille est qu'il n'a ni fils droit, ni fils gauche.

Pour donner une portée plus générale à la fonction demandée, et pour réutiliser des fonctionnalirtés développées dans des exercices précédents, nous nous proposons d'extraire les feuilles d'un arbre sous forme de liste chaînée, laquelle pourra être affichée à l'aide d'une fonction d'impression de liste.

Spécification de l'algorithme

```
FONCTION getLeaves(t : tree) : list
VAR sagldpl, sadldpl : list
DEBUT
  // préconditions et prétraitements standards des cas aux limites
  SI (isEmptyTree(t)) ALORS
    RETOURNER emptyList()
  FINSI
  SI (isSingletonTree(t)) ALORS
    RETOURNER newSingletonList(t→content)
  FINSI
  // sinon :
  sagldpl ← getLeaves(t→sag)
  sadldpl ← getLeaves(t→sad)
  RETOURNER *concat(&sagldpl, &sadldpl)
FIN
```

Exercice 4.3 rechercher itérativement un élément dans un ABOH

Spécification de l'algorithme

```
FONCTION containsElement(t : tree, k : entier) : booléen
DEBUT
  TANTQUE (t ≠ NULL) FAIRE
    SI (t→content = k) ALORS
      RETOURNER vrai
    FINSI
    SI (t→content > k) ALORS
      t ← t→sad
    SINON
      t ← t→sag
    FINSI
  FAIT
  RETOURNER faux
FIN
```

Exercice 4.4 évaluer le caractère ABOH d'un AB

Étude

En terminologie courante, on appelle un ABOH un ABR (Arbre binaire de recherche), en anglais, un BST (*Binary Search Tree*).

L'erreur fréquente consiste à ne vérifier que la condition `fils gauche < parent < fils droit` sur tous les nœuds et non pas `descendants gauches < parent < descendants droits`.

Donnons donc la « mauvaise » et la « bonne » solution à toutes fins utiles.

Spécification de l'algorithme

```
FONCTION isPseudoBinarySearchTree(t : tree ) : booléen
{
  SI (isEmptyTree(t)) ALORS
    RETOURNER vrai
  FINSI
  SI (isSingletonTree(t)) ALORS
    RETOURNER vrai
  FINSI
  SI (isEmptyTree(t→sag) ET isEmptyTree(t→sad)) ALORS
    RETOURNER vrai
  FINSI
  SI (isEmptyTree(t→sad)) ALORS
    RETOURNER (t→sag→content < t→content) ET
     isPseudoBinarySearchTree(t→sag)
  FINSI
  SI (isEmptyTree(t→sag)) ALORS
    RETOURNER (t→sad→content > t→content) ET
     isPseudoBinarySearchTree(t→sad)
  FINSI
  RETOURNER
    (t→sag→content < t→content) ET
    isPseudoBinarySearchTree(t→sag) ET
    (t→sad→content > t→content) ET
    isPseudoBinarySearchTree(t→sad)
}
FONCTION isBinarySearchTree(t : tree) : booléen
DEBUT
  SI (isEmptyTree(t)) ALORS
    RETOURNER vrai
  FINSI
  SI (isSingletonTree(t)) ALORS
    RETOURNER vrai
  FINSI
  SI (isEmptyTree(t→sag) ET isEmptyTree(t→sad)) ALORS
    RETOURNER vrai
  FINSI
  SI (isEmptyTree(t→sad)) ALORS
    RETOURNER (maxOfTree(t→sag) < t→content) ET
```

```
                isBinarySearchTree(t→sag)
  FINSI
  SI (isEmptyTree(t→sag)) ALORS
    RETOURNER (minOfTree(t→sad) > t→content) ET
                isBinarySearchTree(t→sad)
  FINSI
  RETOURNER
    (maxOfTree(t→sag) < t→content) ET isBinarySearchTree(t→sag)
    ET
    (minOfTree(t→sad) > t→content) ET isBinarySearchTree(t→sad)
FIN
// Attention, cette fonction n'est pas définie pour un arbre vide !
FONCTION minOfTree(t : tree) : entier
VAR min, sagMin, sadMin : entier
DEBUT
  min ← t->content
  SI (¬ isEmptyTree(t→sag))ALORS
    sagMin ← minOfTree(t→sag)
    SI (sagMin < min) ALORS
      min ← sagMin
    FINSI
  FINSI
  SI (¬ isEmptyTree(t→sad)) ALORS
    sadMin ← minOfTree(t→sad)
    SI (sadMin < min) ALORS
      min ← sadMin
    FINSI
  FINSI
  RETOURNER min
FIN
// Attention, cette fonction n'est pas définie pour un arbre vide !
FONCTION maxOfTree(t : tree) : entier
VAR max, sagMax, sadMax : entier
DEBUT
  max ← t→content
  SI (¬ isEmptyTree(t→sag)) ALORS
    sagMax ← maxOfTree(t→sag)
    SI (sagMax > max) ALORS
      max ← sagMax
    FINSI
  FINSI
  SI (¬ isEmptyTree(t→sad)) ALORS
    sadMax ← maxOfTree(t→sad)
```

```
    SI (sadMax > max) ALORS
      max ← sadMax
    FINSI
  FINSI
  RETOURNER max
FIN
```

Exercice 4.5 mesurer la hauteur d'un ABOH

Spécification de l'algorithme

```
FONCTION getDepth(t : tree) : entier
VAR sagDepth, sadDepth, maxDepth : entier
DEBUT
  SI (isEmptyTree(t)) ALORS RETOURNER 0
  SINON
    sagDepth ← getDepth(t→sag)
    sadDepth ← getDepth(t→sad)
    SI sagDepth > sadDepth ALORS
      maxDepth ← sagDpeth
    SINON
      maxDepth ← sadDepth
    FINSI
    RETOURNER 1 + maxDepth
  FINSI
FIN
```

Exercice 4.6 évaluer l'équilibre d'un AB

Étude

L'équilibre partiel d'un arbre binaire est déterminé sur la base du différentiel des mesures des hauteurs des fils droit et gauche de la racine.

Par extension, nous proposons une solution qui peut s'appuyer sur différentes mesures, autres que la hauteur, pour évaluer le statut d'équilibre partiel d'un arbre dans une variété élargie de termes.

Le dénominateur commun de toutes les mesures, est qu'elles sont calculées selon un procédé récursif.

Spécification de l'algorithme

```
/**
 * Retourne l'écart en valeur absolue entre
 * la mesure de la partie droite et celle de la partie gauche
 * Le second paramètre utilisé indique la mesure utilisée
 *  1 : masse - 2 : profondeur - 3 : surface
*/
FONCTION getBalance(t : tree, selectedMeasure : entier) : entier
VAR balance : entier
```

```
DEBUT
  SI (isEmptyTree(t) OU isSingletonTree(t)) ALORS
    RETOURNER 1
  FINSI
  balance ← 0
  SI (selectedMeasure = 1) ALORS
    balance ← getMass(t→sag) - getMass(t→sad)
  SINON
    SI (selectedMeasure=2) ALORS
      balance ← getDepth(t→sag) - getDepth(t→sad)
    SINON
      SI (selectedMeasure=3) ALORS
        balance ← getSurface(t→sag) - getSurface(t→sad)
      FINSI
    FINSI
  FINSI
  SI (balance < 0) ALORS
    RETOURNER -balance
  SINON
    RETOURNER balance
  FINSI
FIN
FONCTION getSurface(t : tree) : entier
DEBUT
  SI (isEmptyTree(t)) ALORS
    RETOURNER 0
  FINSI
  SI (isSingletonTree(t)) ALORS
    RETOURNER 1
  SINON
    RETOURNER getSurface(t→sag) + getSurface(t→sad)
  FINSI
FIN
FONCTION getMass(t : tree) : entier
DEBUT
  SI (isEmptyTree(t)) ALORS
    RETOURNER 0
  SINON
    RETOURNER 1 + getMass(t→sag) + getMass(t→sad)
  FINSI
FIN
```

Exercice 4.7 parcourir un AB en largeur et en profondeur

Étude

Deux utilisations classiques de la file et de la pile sont respectivement le parcours en largeur et le parcours en pronfondeur d'un graphe, dont l'arbre est un cas particulier.

Ces deux algorithmes vous préparent à ceux à peine plus compliqués, pour le parcours itératif des graphes généraux

Spécification de l'algorithme

```
FONCTION depthWalkerPrinter(t : tree)
VAR curr : tree ; *s : lStack
DEBUT
  SI (t = NULL) ALORS
    RETOURNER
  FINSI
  s ← newEmptyStack()
  sPush(t, s)
  AFFICHER("dedpthWalkerPrinter : ")
  TANTQUE (¬ isEmptyStack(s)) FAIRE
    sPop(&curr, s)
    AFFICHER(curr→content)
    SI (t→sag ≠ NULL) ALORS
      sPush(t→sag, s)
    FINSI
    SI (t→sad ≠ NULL) ALORS
      sPush(t→sag, s)
    FINSI
  FAIT
FIN
FONCTION widthWalkerPrinter(t : tree)
VAR curr : tree; *q : lQueue
DEBUT
  SI (t = NULL) ALORS
    RETOURNER
  FINSI
  q ← newEmptyQueue()
  qPush(t, q)
  AFFICHER("widthWalkerPrinter : ")
  TANTQUE (¬ isEmptyQueue(q)) FAIRE
    qPop(&curr, q)
    AFFICHER(curr→content)
    SI (t→sag ≠ NULL) ALORS
      qPush(t→sag, q)
```

```
    FINSI
    SI (t→sad ≠ NULL) ALORS
      qPush(t→sag, q)
    FINSI
  FAIT
FIN
```

Exercice 4.8 effectuer un calcul complexe sur un arbre

Étude

L'ordre de parcours de l'arbre importe peu, de même que la méthode (itérative ou récursive), mais il est nécessaire de parcourir l'intégralité de l'arbre pour calculer quatre grandeurs : les sommes et dénombrements respectifs des éléments respectivement positifs et négatifs de l'arbre.

- *Spécification de l'algorithme*

Fonction principale simple

```
/**
 * Retourne la moyenne des éléments positifs
 * et celle des éléments négatifs d'un arbre d'entiers
 * Si l'arbre est vide, 'ppo_av' et 'pne_av' sont fixés à -1,
 * Sinon retour respectif des deux moyennes en valeur absolue
 */
FONCTION signedAveragesMain(t : tree, *ppo_av : réel, *pne_av : réel)
VAR po_sum, po_count, ne_sum, ne_count : entier
DEBUT
  signedAverages(t, &po_sum, &po_count, &ne_sum, &ne_count)
  SI po_count = 0 ALORS
    *ppo_av ← -1
  SINON
    *ppo_av ← po_sum / po_count
  FINSI
  SI ne_count = 0 ALORS
    *pne_av ← -1
  SINON
    *pne_av ← ne_sum / ne_count
  FINSI
FIN
```

Fonction secondaire récursive

```
/**
 * Fonction (privée) récursive qui calcule respectivement
 * les sommes et comptes de positifs et négatifs
 * ppo_sum : somme des positifs, ppo_count : nombre de positifs,
 * pne_sum : somme des négatifs, pne_count : nombre de négatifs
```

```
 */
FONCTION signedAverages(t : tree, *ppo_sum : entier,
  *ppo_count : entier, *pne_sum : entier, *pne_count : entier)
VAR sag_po_sum, sag_po_count, sag_ne_sum, sag_ne_count : entier
VAR sad_po_sum, sad_po_count, sad_ne_sum, sad_ne_count : entier
DEBUT
  /// Condition d'arrêt
  SI (t = NULL) ALORS
    *ppo_sum ← 0
    *ppo_count ← 0
    *pne_sum ← 0
    *pne_count ← 0
    RETOURNER
  FINSI
  signedAverages(t→sag,
    &sag_po_sum, &sag_po_count, &sag_ne_sum, &sag_ne_count)
  signedAverages(t→sad,
    &sad_po_sum, &sad_po_count, &sad_ne_sum, &sad_ne_count)
  *ppo_sum ← sag_po_sum + sad_po_sum
  *ppo_count ← sag_po_count + sad_po_count
  *pne_sum ← sag_ne_sum + sad_ne_sum
  *pne_count ←sag_ne_count + sad_ne_count
  SI (t→content < 0) ALORS
    (*pne_sum) ← (*pne_sum) - t→content
    (*pne_count) ← (*pne_count)+1
  SINON
    (*ppo_sum) ← (*ppo_sum) + t→content
    (*ppo_count) ← (*ppo_count)+1
  FINSI
FIN
```

- *Réalisation en C*

Une fonction principale réalise et retourne les deux moyennes en s'appuyant sur une fonction récursive secondaire qui calcule les quatre grandeurs.

Fonction principale simple

```
void signedAveragesMain(tree t, double *ppo_av, double *pne_av)
{
  int po_sum, po_count, ne_sum, ne_count;
  signedAverages(t, &po_sum, &po_count, &ne_sum, &ne_count);
  *ppo_av = po_count == 0 ? -1 : (double) po_sum / (double) po_count;
  *pne_av = ne_count == 0 ? -1 : (double) ne_sum / (double) ne_count;
}
```

Fonction secondaire récursive

```
void signedAverages(tree t,
  int *ppo_sum, int *ppo_count, int *pne_sum, int *pne_count)
{
  /// Condition d'arrêt
  if (t == NULL)
  {
    *ppo_sum = 0;
    *ppo_count = 0;
    *pne_sum = 0;
    *pne_count = 0;
    return;
  }
  /// Sinon :
  int sag_po_sum, sag_po_count, sag_ne_sum, sag_ne_count;
  int sad_po_sum, sad_po_count, sad_ne_sum, sad_ne_count;
  signedAverages(t->sag,
    &sag_po_sum, &sag_po_count, &sag_ne_sum, &sag_ne_count);
  signedAverages(t->sad,
    &sad_po_sum, &sad_po_count, &sad_ne_sum, &sad_ne_count);
  *ppo_sum = sag_po_sum + sad_po_sum;
  *ppo_count = sag_po_count + sad_po_count;
  *pne_sum = sag_ne_sum + sad_ne_sum;
  *pne_count = sag_ne_count + sad_ne_count;
  if (t->content < 0)
  {
    (*pne_sum) -= t->content;
    (*pne_count)++;
  }
  else
  {
    (*ppo_sum) += t->content;
    (*ppo_count)++;
  }
}
```

Pour tester

```
/**
 * Retourne un arbre parfait, de racine de valeur 'n' et profondeur 'p',
 * dont chaque sag a pour valeur celle de son parent -1,
 * et dont chaque sad a pour valeur celle de son parent +1
 */
tree treeAveragesTestTree(int n, unsigned p)
```

```
{
  /// Condition d'arrêt
  tree t = newSingletonTree(n);
  if (p == 0) return t;
  /// Sinon
  p--;
  t->sag = treeAveragesTestTree(n - 1, p);
  t->sad = treeAveragesTestTree(n + 1, p);
  return t;
}
void testEx2()
{
  printf(">> tree averages\n");
  tree t = treeAveragesTestTree(0, 10);
  simplePrintTreeGraph(t, "ttest");
  double po_av, ne_av;
  signedAveragesMain(t, &po_av, &ne_av);
  int po_sum, po_count, ne_sum, ne_count;
  signedAverages(t, &po_sum, &po_count, &ne_sum, &ne_count);
  signedAveragesMain(t, &po_av, &ne_av);
  printf("ttest positives average : %.4f (%d/%d) -",
    po_av, po_sum, po_count);
  printf("negatives average : %.4f (%d/%d)\n",
    ne_av, ne_sum, ne_count);
}
```

Exercice 4.9 extraire une liste d'éléments d'un arbre

Étude

- *Analyse*

Il s'agit d'extraire d'un arbre tous les fils premiers avec leur parent pour les restituer sous forme de liste chaînée. Rien n'indique qu'il s'agit d'extraire une liste sans répétitions (voir exemple d'arbre avec rotation à trois éléments ci-dessous, dans la section « réalisation C/pour tester »).

Nous nous proposons de réaliser un algorithme basé sur le calcul du PGCD(parent, enfant) à l'aide de l'algorithme d'Euclide, espérant ainsi obtenir un bonus de la part du correcteur : le PGCD doit valoir 1 pour caractériser la primalité respective du parent et de l'enfant. Mais la simple vérification de la nullité du modulo de l'un par l'autre est suffisante pour déterminer cette propriété.

Peu importe le type de parcours choisi (pré, in, ou post fixe), l'algorithme est naturellement récursif et basé sur la fusion de proche en proche des listes d'enfants premiers de chaque sous-arbre : les parents des feuilles retournent les premières listes singletons, lesquelles sont enrichies et concaténées (l'opération de fusion) au fur et à mesure du dépilement de la pile d'appel, et finalement fusionnées en une seule et unique liste au moment du dépilement final de l'appel initial de la fonction.

• *Méthodes de construction*

Un arbre évident dont tous les fils sont premiers avec leur parent est l'arbre qui ne contient que des nombres premiers. Une autre façon de construire un tel arbre est d'effectuer la répétition d'un motif appropriée de trois éléments premiers entre eux, par exemple 4, 9 et 25 : par exemple, 4 prend la place de racine, 9 celle de fils gauche, 25 celle de fils droit, puis 4 prend la place de fils gauche de 9 et de 25, et 25 celle de fils droit de 9 et 9 celle de fils droit de 25, et ainsi de suite...

Il existe une multitude de façons de procéder à la construction d'un arbre qui possède cette propriété de primalité relative parent/enfant.

Cependant, l'exactitude mathématique invite à préciser qu'en réalité, il n'existe aucune façon de construire un tel arbre qui n'existe pas. En effet, la racine de l'arbre ne peut être première avec son parent qui n'existe pas.

Spécification de l'algorithme

```
/**
 * Retourne la liste chaînée des éléments d'un arbre d'entiers,
 * premiers avec leur parent
 */
FONCTION getPrimeChildren(t : tree) : list
VAR sagprimechildren, sadprimechildren, primechild : list
DEBUT
  SI (t = NULL) ALORS
    RETOURNER NULL
  FINSI
  sagprimechildren ← NULL
  sadprimechildren ← NULL
  SI (t→sag ≠ NULL) ALORS
    sagprimechildren ← getPrimeChildren(t→sag)
    SI (gcd(t→content, t→sag→content) = 1) ALORS
      primechild ← newSingletonList(t→sag→content)
      SI (primechild ≠ NULL) ALORS
        primechild→succ ← sagprimechildren
        sagprimechildren ← primechild
      FINSI
    FINSI
  FINSI
  SI (t→sad ≠ NULL) ALORS
    sadprimechildren ← getPrimeChildren(t→sad)
    SI (gcd(t→content, t→sad→content) = 1) ALORS
      primechild ← newSingletonList(t→sad→content)
      SI (primechild ≠ NULL) ALORS
        primechild→succ ← sadprimechildren
        sadprimechildren ← primechild
      FINSI
```

```
    FINSI
  FINSI
  RETOURNER *concat(&sagprimechildren, &sadprimechildren)
FIN
/** Algorithme d'Euclide accessible dans les Elemens **/
FONCTION gcd(numerator : entier, denominator : entier) : entier
DEBUT
  SI (denominator = 0) ALORS
    RETOURNER numerator
  SINON
    RETOURNER gcd(denominator, numerator % denominator)
  FINSI
FIN
```

Réalisation en C

```
list getPrimeChildren(tree t)
{
  if (t == NULL) return NULL;
  list sagprimechildren = NULL;
  list sadprimechildren = NULL;
  list primechild;
  if (t->sag != NULL)
  {
    sagprimechildren = getPrimeChildren(t->sag);
    if (gcd(t->content, t->sag->content) == 1)
    {
      primechild = newSingletonList(t->sag->content);
      if (primechild != NULL)
      {
        primechild->succ = sagprimechildren;
        sagprimechildren = primechild;
      }
    }
  }
  if (t->sad != NULL)
  {
    sadprimechildren = getPrimeChildren(t->sad);
    if (gcd(t->content, t->sad->content) == 1)
    {
      primechild = newSingletonList(t->sad->content);
      if (primechild != NULL)
      {
        primechild->succ = sadprimechildren;
```

```
        sadprimechildren = primechild;
      }
    }
  }
  return *concat(&sagprimechildren, &sadprimechildren);
}
unsigned gcd(unsigned numerator, unsigned denominator)
{
  if (denominator == 0) return numerator;
  else return gcd(denominator, numerator % denominator);
}
```

Pour tester

```
/**
 * Retourne un arbre parfait de profondeur 'p',
 * et dont les 3 valeurs v1, v2, v3 initient la récurrence suivante :
 *   v1 est la valeur de la racine,
 *   v2 celle de son fils gauche, v3 celle de son fils droit,
 *   si un nœud a pour valeur v(i),
 *     alors son fils gauche a pour valeur v((i + 1) mod 3),
 *     et son fils droit v((i + 2) mod 3)
 */
tree primeChildrenTestTree(unsigned p,
  unsigned v1, unsigned v2, unsigned v3)
{
  /// Condition d'arrêt
  tree t = newSingletonTree(v1);
  if (p == 0) return t;
  /// Sinon
  p--;
  t->sag = primeChildrenTestTree(p, v2, v3, v1);
  t->sad = primeChildrenTestTree(p, v3, v1, v2);
  return t;
}
void testPrimeChildren()
{
  printf(">> tree prime children\n");
  tree p = primeChildrenTestTree(3, 1, 2, 3);
  simplePrintTreeGraph(p, "ptest (123)");
  list primechildren = getPrimeChildren(p);
  printListGraph(primechildren, "primechildren (123)");
  p = primeChildrenTestTree(3, 1, 2, 1);
  simplePrintTreeGraph(p, "ptest (121)");
```

```
  primechildren = getPrimeChildren(p);
  printListGraph(primechildren, "primechildren (121)");
  p = primeChildrenTestTree(3, 2, 2, 2);
  simplePrintTreeGraph(p, "ptest (222)");
  primechildren = getPrimeChildren(p);
  printListGraph(primechildren, "primechildren (222)");
}
```

Exercice 4.10 produire des coupes transversales d'un arbre

Spécification de l'algorithme

```
/**
 * Retourne sous forme de liste, la coupe tranversale d'un arbre
 * du niveau de pronfondeur 'depth'
 */
FONCTION getLevelList(t : tree, depth : entier) : list
VAR sagll, sadll : list
DEBUT
  SI (getDepth(t) < depth) ALORS
    RETOURNER NULL
  FINSI
  SI (depth = 1) ALORS
    SI  (t = NULL) ALORS
      RETOURNER NULL
    SINON
      RETOURNER newSingletonList(t→content)
    FINSI
  SINON
    depth ← depth-1
    sagll ← getLevelList(t→sag, depth)
    sadll ← getLevelList(t→sad, depth)
    RETOURNER *concat(&sagll, &sadll)
  FINSI
FIN
```

Corrigés des problèmes

Problème 4.1 le triangle chinois de Pascal

Analyse

- *Traduction du chinois vers l'arabe/latin*

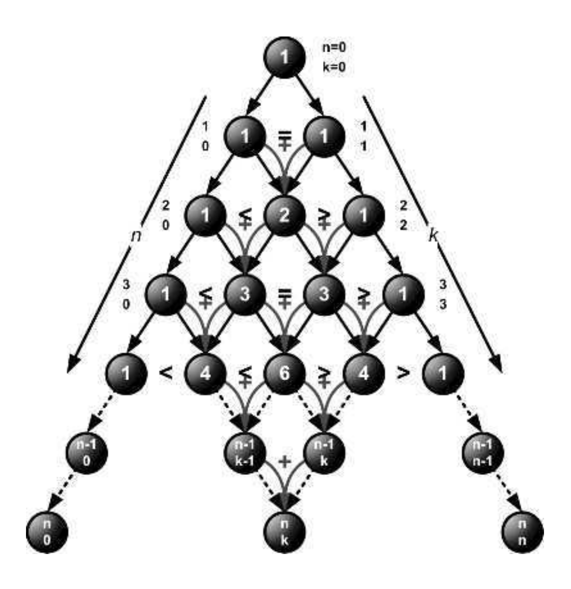


Figure 4.24

- *Ce qui est demandé*

De passer d'une structure qui n'est pas un arbre mais s'en approche à un format d'arbre. Pour cela, on transforme les filiations croisées en filiations simples en supprimant les relations parent-enfant sur la gauche, sauf pour le côté droit de l'arbre.

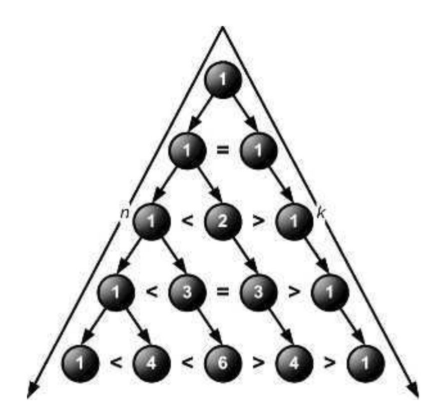


Figure 4.25

Stratégie de résolution

- *Utilisation de la formule de récurrence de Pascal*

Le procédé de construction d'un arbre est naturellement récursif et en particulier celui-ci.

La formule de Pascal est une récurrence qu'il convient d'exploiter pour éviter de recalculer pour chaque nœud de l'arbre, une expression qui comprend trois factorielles dont on sait le coût prohibitif.

Ne pas exploiter cette récurrence est une véritable erreur professionnelle pour un informaticien... C'est ce qui a justifié un bonus supplémentaire pour les quelques-unes et quelques-uns qui y ont pensé.

- *Approche récursive ou itérative ?*

Comme on supprime les filiations gauches, il faut qu'un « parent gauche » puisse passer la valeur du « parent droit », non relié, à tout nouveau « fils droit » qu'il crée.

Si on reste sur une vision « arbre », ça peut devenir compliqué, en particulier en cherchant une méthode récursive.

Or, si l'on constate que l'arbre n'est rien d'autre qu'une liste de listes, la première liste étant la branche de droite (1, 1, 1, ...), puis la branche (1, 2, 3, ...), puis (1, 3, 6, ...), alors le problème apparaît plus simple et se prête naturellement à une procédure itérative à deux boucles.

La figure 4.26 (la même sous une perspective à peine différente) devrait vous aider à vous le représenter.

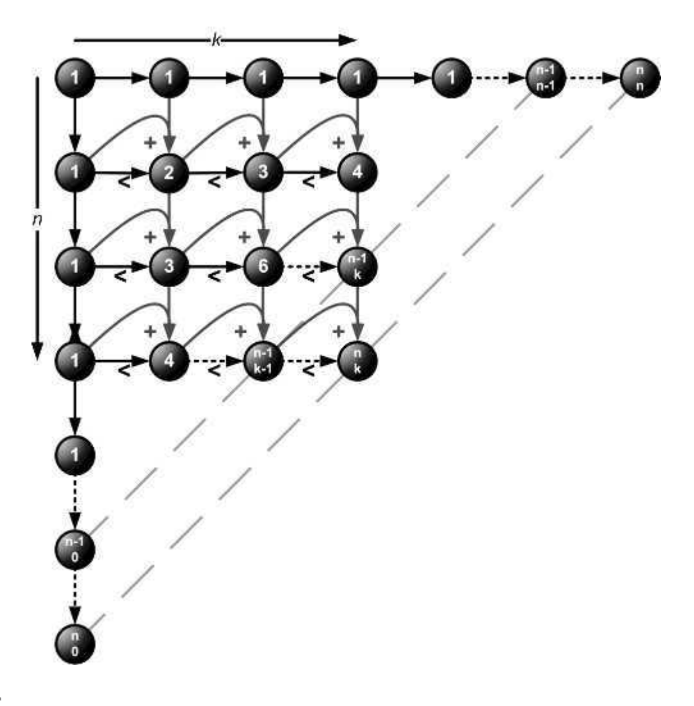


Figure 4.26

Spécification de l'algorithme

```
FONCTION buildPascalTriangle(*pt : tree, n : entier)
VAR i, j : entier; root, curr, currline, prev, prevline : tree
DEBUT
  *pt ← newSingletonTree(1)
  root ← *pt
  SI (n = 0) ALORS
    RETOURNER
  FINSI
  j ← n + 1
  currLine ← root
  curr ← currLine
  TANTQUE (j > 0) FAIRE
    curr→sad ← newSingletonTree(1)
    curr ← curr→sad
    j ← j-1
  FAIT
  prevLine ← root
  prev ← prevLine
  i ← n
  TANTQUE (i > 0) FAIRE
    prevLine→sag ← newSingletonTree(1)
    currLine ← prevLine→sag
    curr ← currLine
    j ← i
    TANTQUE (j > 0) FAIRE
      prev ← prev→sad
      curr→sad ← newSingletonTree(curr→content + prev→content)
      curr ← curr→sad
      j ← j-1
    FAIT
    prevLine ← currLine
    prev ← prevLine
    i ← i-1
  FAIT
FIN
```

Réalisation en C

```c
void buildPascalTriangle(tree* pt, unsigned n)
{
  tree root = *pt = newSingletonTree(1);
  if (n == 0) return;
  unsigned j = n + 1;    /// itérateur de colonnes (indices de 0 à n)
```

```
    tree currLine = root;   /// ligne de sad->sad->... courante
    tree curr = currLine;   /// fils sad en création courant
    /// on commence par créer toute la branche droite de l'arbre
    while (j-- > 0)
    {
        curr->sad = newSingletonTree(1);
        curr = curr->sad;
    }
    /// le travail s'effectue ensuite ligne après ligne,
    /// avec une longueur qui se réduit à chaque étape
    tree prevLine = root;      /// ligne de sad->sad->... précédente
    tree prev = prevLine;
    unsigned i = n;
    while (i-- > 0)
    {
      /// passage à la ligne suivante : d'abord créer sa tête
      currLine = prevLine->sag = newSingletonTree(1);
      curr = currLine;
      unsigned j = i;
      /// on co-itère la ligne précédente avec la ligne en cours
      /// de création pour profiter de la formule de Pascal
      while (j-- > 0)
      {
        prev = prev->sad;
        curr->sad = newSingletonTree(curr->content + prev->content);
        /// -> clé de performance de l'algo
        curr = curr->sad;
      }
      prevLine = currLine;
      prev = prevLine;
    }
  }
```

Complexité algorithmique comparée

Pour que le gain entre le calcul de toutes factorielles, et l'exploitation de la formule de récurrence de Pascal ne soit pas seulement abstrait, évaluons la complexité algorithmique comparée entre les deux approches en fonction d'un paramètre n, qui représente le niveau de profondeur de l'arbre, avec premier niveau pour $n = 0$.

Pour calculer une factorielle d'un entier n, il faut réaliser $n – 1$ produits avec affectation de chaque résultat intermédiaire dans une variable temporaire.

Notons $c(f(n))$, le coût du calcul d'une fonction $f(n)$: $c\,(n!) = 2\,(n - 1)$

Pour calculer une combinaison, nous avons trois factorielles, un produit et une division, trois affectations :

$$c\left(C_n^k\right) = c\left(\frac{n!}{k!\,(n-k)!}\right) = c\,(n!) + c\,(k!) + c\,((n-k)!) + 4 = 2\,(2n-1)$$

La complexité est donc linéaire à comparer avec une complexité en temps constant (deux opérations, une somme et une affectation) pour un calcul par la formule de récurrence de Pascal.
Si l'on passe à l'échelle de la ligne de profondeur n, il y a $n + 1$ termes à calculer, et donc le coût en opérations pour une ligne du triangle de Pascal est $2\,(2n - 1)\,(n + 1)$ avec la mauvaise méthode et $2\,(n + 1)$ avec la bonne.
À l'échelle d'un triangle de profondeur n, nous obtenons donc un coût de :

$$\sum_{i=1}^{n} 2\,(2i-1)\,(i+1)$$

avec la mauvaise méthode, et un coût de $\sum_{i=1}^{n} 2\,(i+1) = (n+1)(n+2)$, avec la bonne.

Problème 4.2 interlude Syracusien

Fonction de retour du prédécesseur

- *Le principe*

Il s'agit de vérifier dans un premier temps que *(n – 1)* est divisible par *3*. Cela se vérifie facilement à l'aide du test `(n - 1) % 3 = 0`, soit `m % 3 = 0` avec `m = n - 1`.
Mais ce n'est pas suffisant, puisqu'il faut par ailleurs que *m/3* soit impair, car sinon, le terme trouvé aurait pour successeur sa moitié et non pas *n*, c'est-à-dire que son successeur serait *m/6* et non pas *n*.
L'imparité de *m/3* peut se vérifier à l'aide de `(m/3) % 2 = 1` ou `≠ 0`, mais si le résultat invalide cette imparité, une division aura été effectuée pour rien.
Il suffit de constater que si *m* est à la fois multiple de *3*, et multiple de *2*, alors il est multiple de *6*, et donc `m % 6 = 0`.
La condition d'imparité de *m/3* peut donc être vérifiée par `m % 6 ≠ 0`. Si les deux conditions sont validées, alors et seulement alors, on calcule et on retourne *m/3*.
Cette optimisation est importante, car cette fonction utilitaire est appelée à chaque étape de la récursion ci-après, et que son coût d'exécution représente un coût élémentaire facteur direct du coût global de l'exécution de l'algorithme principal.

- *Réalisation*

```
/** Retourne le prédécesseur impair de n tel que n = prec * 3 + 1
s'il existe 0 sinon **/
FONCTION precInf(n : entier) : entier
VAR m : entier
DEBUT
  m ← n - 1
```

```
    SI ((m % 3) ≠ 0) ALORS
      RETOURNER 0
    FINSI
    SI ((m % 6) = 0) ALORS
      RETOURNER 0
    FINSI
    RETOURNER m / 3
FIN
```

- *Optimisation*

Une solution plus efficace peut être trouvée moyennant un petit raisonnement d'arithmétique modulaire :

$$\forall n \in \mathbb{N} : \exists p \in \mathbb{N}^* : n = 3p + 1 \wedge \text{impair}\,[p]$$
$$\Leftrightarrow \exists ! k \in \mathbb{N} : n = 3\,(2k + 1) + 1 = 6k + 4$$
$$\Leftrightarrow n \equiv 4\,(6)$$

Et donc la fonction se réduit à :

```
FONCTION precInf(n : entier) : entier
DEBUT
    SI n%6 = 4 ALORS
      RETOURNER (n-1)/3
    SINON
      RETOURNER 0
    FINSI
FIN
```

Cette réalisation équivalente économise de nombreuses opérations élémentaires (soustractions inutiles, modulos intermédiaires, négations logiques) à rapporter au fait que l'appel de cette fonction utilitaire est systématique au fur et à mesure de la récursion.

Fonction récursive de construction de l'arbre

```
/**
 * Retourne l'arbre de Syracuse (tête valant 1)
 * jusqu'à la profondeur maxDepth
 */
FONCTION syracuseTreeGrowth(pt : ptree, maxDepth : entier)
VAR n, prec_inf_n : entier; t : tree
DEBUT
    SI (maxDepth = 0) ALORS
      RETOURNER
    FINSI
    t ← *pt
    SI (t = NULL) ALORS
```

```
    t ← newSingletonTree(1)
    *pt ← t
  FINSI
  n ← t→content
  maxDepth ← maxDepth-1
  t→sad ← newSingletonTree(2 * n)
  syracuseTreeGrowth(&(t→sad), maxDepth)
  prec_inf_n ← precInf(n)
  SI (prec_inf_n > 1) ALORS
    t→sag ← newSingletonTree(prec_inf_n)
    syracuseTreeGrowth(&(t→sag), maxDepth)
  FINSI
FIN
```

AUTOMATES 5

RAPPELS DE COURS

5.1 HISTORIQUE

Alan Turing a décrit à 1936 un modèle de « machine idéale » auquel il a laissé son nom. D'autres modèles furent proposés, qui sont tous équivalents à la **machine de Turing**. Church a démontré dans sa thèse que ces modèles sont les plus généraux possibles.

Ces travaux amenèrent à distinguer entre les fonctions qui sont calculables en théorie de celles qui ne le seraient même pas. (Toute fonction calculable peut être calculée par la machine de Turing en temps (nombre d'opérations) fini).

Un modèle plus simple mais fort utile est celui d'un **automate fini**. Un automate fini est une machine abstraite constituée d'**états** et de **transitions**. Cette machine est destinée à traiter des **mots** fournis en entrée : l'automate passe d'état en état, suivant les transitions, à la lecture de chaque lettre de l'entrée. L'automate est dit « fini » car il possède un nombre fini d'états distincts : il ne dispose donc que d'une mémoire bornée, indépendamment de la taille de la donnée sur laquelle on effectue les calculs.

Un automate fini forme un graphe orienté étiqueté, dont les états sont les sommets et les transitions les arcs étiquetés.

Les automates finis fournissent un outil de construction d'algorithmes particulièrement simple.

Il existe plusieurs types de machines à états finis. Les « **accepteurs** » produisent en sortie une réponse « oui » ou « non », c'est-à-dire qu'ils acceptent (oui) ou rejettent (non) l'entrée. Les systèmes de reconnaissance classent l'entrée par catégorie. Les capteurs produisent un certain résultat en fonction de l'entrée.

Les automates finis peuvent caractériser des langages (c'est-à-dire des ensembles de mots) finis (le cas standard), des langages de mots infinis (automates de Rabin, automates de Büchi), ou encore divers types d'arbres (automates d'arbres).

Les machines à états finis se rencontrent également dans les circuits digitaux, où l'entrée, l'état et le résultat sont des vecteurs de bits de taille fixe (machines de Moore et machines de Mealy). Dans les machines de Mealy, les actions (sorties) sont liées aux transitions, tandis que dans les machines de Moore, les actions sont liées aux états.

5.2 QUELQUES DÉFINITIONS

Pour définir un automate fini, on a besoin des éléments suivants :

- **Un alphabet** A qui est un ensemble fini d'objets qu'on appelle « lettres » ou « caractères » mais qui peuvent, selon le cas, être de n'importe quel genre (instructions machine, état civil d'une personne, type d'objet géométrique etc.)
- Des **mots** sont des séquences ordonnées de caractères de l'alphabet. Dans nos applications, on aura affaire qu'à des mots de longueur finie, mais il n'est pas interdit d'utiliser des mots de longueur infinie (automates de Rabin, automates de Büchi). La longueur d'un mot est le nombre de lettres le constituant.
  Exemple : $w = abacda$ – mot de longueur six sur l'alphabet latin (ou bien sur l'alphabet $\{a, b, c, d\}$).
- **Un mot vide** est noté par ε ou par 1. C'est le seul mot de longueur nulle.
- On note A^* **l'ensemble** de tous les mots sur l'alphabet A, y compris le mot vide.
- Un automate fini sur l'alphabet A est donné par un quadruplet (Q, I, T, E) où :
  - Q est un ensemble **fini** d'états de l'automate ;
  - $I \subset Q$ est un ensemble d'états initiaux ;
  - $T \subset Q$est un ensemble d'états terminaux ;
  - $E \subset Q \times \left(A \cup \{\varepsilon\}\right) \times Q$ est un ensemble de triplets $(p.a.q)$ appelés les flèches ou les transitions de l'automate.

Remarque

Nous allons temporairement oublier la notion du mot vide, jusqu'à l'introduction explicite d'automates asynchrones. Dans tous les exemples que nous allons traiter jusque ce point-là, on pourra donc définir l'ensemble E comme faisant partie de $Q \times A \times Q$.

5.3 L'INTERPRÉTATION INTUITIVE

À chaque instant l'automate se trouve dans l'un des états p de l'ensemble d'états Q. Il lit alors l'une des lettres a de l'alphabet A (la lettre suivant du mot qui vient à l'entrée) et passe dans un état q tel que $(p.a.q)$ soit une flèche.

Exemple

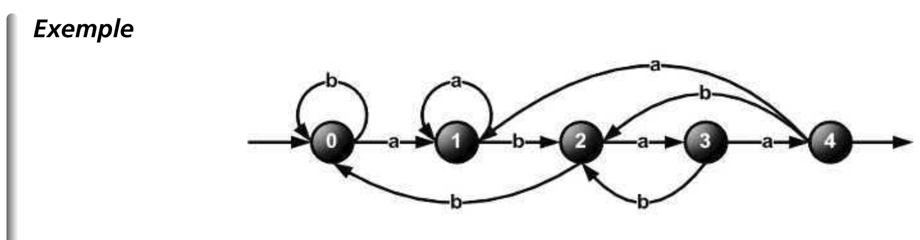


Figure 5.1

$$A = \{a, b\};\ Q = \{0,1,2,3,4\};\ I = \{0\};\ T = \{4\}$$

La lecture du mot w se termine dans l'état 4 ssi w se termine par $abaa$.
Le même automate peut être représenté par une **table de transitions** :

Tableau 5.1

	État	État résultant	
		en lisant a	en lisant b
(entrée)	0	1	0
	1	1	2
	2	3	0
	3	4	2
(sortie)	4	1	2

Remarque

Si vous regardez cet automate de près, vous verrez que nous avons libellé les états de telle façon que l'automate se trouve dans l'état numéro i ssi le plus long **suffixe du mot déjà lu** qui est en même temps un **préfixe de** $abaa$**,** est de **longueur** i.

On peut montrer que cet automate réalise de façon optimale l'algorithme de recherche de la séquence $abaa$ dans un texte : il résout pour ce mot l'algorithme du « *string matching* » utilisé par exemple dans les éditeurs de textes.

La lecture

L'automate prend un mot (constitué de symboles de son alphabet) en entrée et démarre dans son ou ses état(s) initial(-aux). Il parcourt ensuite le mot de gauche à droite : si l'automate se trouve dans un état p et le symbole à lire est a, alors s'il existe(nt) un ou des état(s) q_i tels que la transition $(p.a.q_i)$ existe, il passe aux états q_i, et de suite. (S'il y a plus d'un état q_i, on considère chacun d'eux séparément). La lecture se termine soit si la lettre suivante ne peut pas être lue (il n'y a pas de transition y correspondant), soit si on a atteint la fin du mot. Si, à la fin de la lecture, l'automate est dans un état final (accepteur), on dit qu'il accepte l'entrée ou qu'il la reconnaît. Autrement, on dit qu'il la rejette.

Notation

Pour un mot $w \in A^*$ on note $p \xrightarrow{w} q$ s'il existe un chemin de l'état p à l'état q obtenu en lisant w.

Définitions formelles

$\forall a \in A$, on a $p \xrightarrow{a} q$ ssi il existe une flèche $(p.a.q) \in E$.

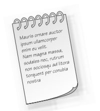
Ensuite, pour $u \in A^*$ et $a \in A$ on a $p \xrightarrow{ua} q$ ssi $\exists r \in Q$ tq $p \xrightarrow{u} r$ et $r \xrightarrow{a} q$.

Quand on a $p \xrightarrow{w} q$ on va dire qu'il existe un chemin de p à q d'étiquette w.

On peut facilement voir (par récurrence) que pour $u, v \in A^*$ on a :

$$p \xrightarrow{uv} q \text{ ssi } \exists r \in Q \text{ tq } p \xrightarrow{u} r \text{ et } r \xrightarrow{v} q.$$

Un mot $w \in A^*$ est donc reconnu par l'automate fini s'il existe un chemin $i \xrightarrow{uv} t$ où $i \in I$ et $t \in T$ – c.-à-d. qu'en partant d'un état initial et qu'en lisant le mot w, on atteint un état terminal à la fin de la lecture.



Le **langage** L **reconnu** par l'automate fini est l'ensemble de tous les mots reconnus.

Exemple d'exercice type

Construire un automate fini reconnaissant les entiers écrits en base deux et divisibles par trois.

Solution

Ajout d'un 0 à la fin d'un nombre binaire le multiplie par 2.
Ajout d'un 1 à la fin d'un nombre binaire le multiplie par 2 et lui ajoute 1.
Marquant les états par le reste de la division entière par 3 :

Tableau 5.2

N	*N* mod3	ajout d'un 0 à la fin	reste	ajout d'un 1 à la fin	reste
$3n$	0	$6n$	0	$6n + 1 = 3 \times 2n + 1$	1
$3n + 1$	1	$6n + 2 = 3 \times 2n + 2$	2	$6n + 3 = 3 \times (2n + 1)$	0
$3n + 2$	2	$6n + 4 = 3 \times (2n + 1) + 1$	1	$6n + 5 = 3 \times (2n + 1) + 2$	2

Notons les états par le reste de la division entière par 3. On obtient :

Tableau 5.3

État	État résultant	
	0	1
0	0	1
1	2	0
2	1	2

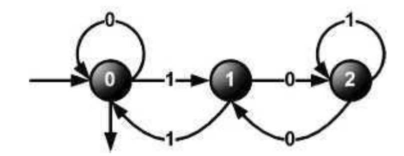


Figure 5.2

En fait, cet automate est bon pour reconnaître des nombres en écriture binaire avec n'importe quel reste de la division entière par 3, au prix de choisir quel état est terminal. L'état terminal 0 correspond au reste 0 (divisibilité par 3) ; l'état terminal 1, au reste 1 ; l'état terminal 2, au reste 2.

5.3.1 Automates déterministes

On distingue des automates finis déterministes et non déterministes. L'automate que nous avons montré précédemment est déterministe.

Voici l'exemple d'un automate ***non déterministe*** :

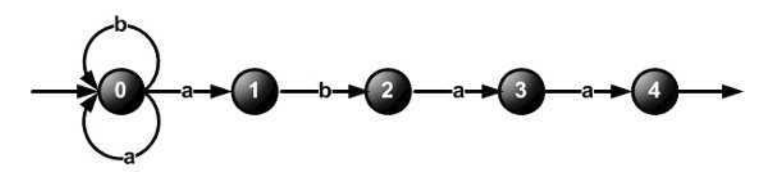


Figure 5.3 **Cet automate reconnaît tous les mots qui se terminent par abaa. Il est très facile à construire.**

Le fait qu'il n'est pas déterministe se manifeste en ce que, en lisant ***a*** à partir de l'état 0, on peut aller et à l'état 0, et à l'état 1, ce qui ne serait pas possible pour un automate déterministe.

On va montrer dans ce qui suit que l'on peut toujours construire, à partir d'un automate fini quelconque, un autre automate qui est déterministe et qui lui est équivalent (reconnaît le même langage).

Automate déterministe, définition

L'automate $C = (Q, I, T, E)$ est déterministe ssi pour $\forall p \in Q$ et $\forall a \in A$ il existe au plus un état $q \in Q$ tq $(p.a.q)$, et qu'il y ait un seul état initial : $I = \{i\}$.

On peut donc caractériser un automate déterministe C par un quadruplet (Q, i, T, E) où i est l'état initial.

Si la transition $(p.a.q)$ existe, on notera $p.a = q$**.** Si elle n'existe pas, on conviendra que $p.a$n'a pas de valeur.

On peut facilement vérifier par récurrence sur la longueur des mots que pour un automate déterministe C**,** pour $\forall p \in Q$ et $\forall$ mot $u \in A^*$, il existe au plus un état $q \in Q$ tq $p \xrightarrow{u} q$.

On notera alors $p.u = q$en convenant que $p.u$ n'est pas défini quand il n'existe pas de tel état q; et on aura pour deux mots $u, v \in A^*$:

$$(p.u).v = p.(u.v)$$

Déterminisation

Soit $C = (Q, I, T, E)$ un automate fini.

Notons Π l'ensemble des parties de Q. C'est un ensemble fini puisque si Q a n éléments, Π en a 2^n.

Un automate déterministe D correspondant à C prend :

- comme ensemble d'états un sous-ensemble P de l'ensemble Π des parties de Q,

- comme unique état initial l'ensemble I des états initiaux de C,
- comme ensemble d'états terminaux l'ensemble $\varsigma = \{u \in P | u \cap T = \emptyset\}$ des parties de Q qui contiennent au moins un état terminal de C,
- comme flèches l'ensemble des $(u.a.v)$ où $u \in P$ et v est l'ensemble de tous les états $q \in Q$ tels qu'il existe un état p dans u avec $(p.a.q) \in E$.

Cet algorithme formel s'explique le plus facilement sur un exemple.

Exemple de déterminisation

Prenons l'automate non déterministe précédent :

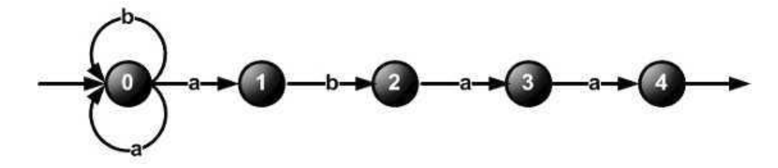


Figure 5.4

- *Déterminisation*

Tableau 5.4

	État	États résultant	
		en lisant a	en lisant b
(entrée)	(0)	(0,1)	(0)
	(0,1)	(0,1)	(0,2)
	(0,2)	(0,1,3)	(0)
	(0,1,3)	(0,1,4)	(0,2)
(sortie)	(0,1,4)	(0,1)	(0,2)

- *Explications*

On commence par l'état initial qui dans ce cas particulier coïncide avec l'état initial de l'automate non déterministe (0), car notre automate non déterministe n'en a qu'un seul. (S'il en avait plusieurs, on les aurait regroupés pour former l'état initial de l'automate déterministe).

À partir de cet état (0), en a on va vers les états (0), (1) de l'automate initial. Pour l'automate déterministe, on les regroupe dans l'état qu'on appelle (0,1). En b, on va vers l'état (0) de l'automate initial. Donc, (0) est lui aussi un état de l'automate déterminisé.

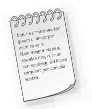
Chaque fois qu'on obtient un état de l'automate déterminisé, on le met dans la colonne de gauche pour agir sur cet état par les lettres de l'alphabet. Ainsi on obtient la deuxième ligne, où figure un nouvel état (0,2). On procède ainsi tant qu'il y reste des états non traités.

Les états finaux sont ceux qui contiennent un état final de l'automate initial. Dans notre cas, il n'y en a qu'un : (0,1,4).

Figure 5.5 **Automate déterminisé**

Voici l'automate déterminisé :

La démonstration montrant que et C reconnaissent le même langage se fait par récurrence.



Un automate déterministe peut être ou ne pas être **complet.**

Automate déterministe complet (définition)

Pour $\forall u \in P$ et $\forall a \in A$, $\exists v \in P$ tel que $u.a = v$.

C'est-à-dire que de chaque état sortent des flèches avec toutes les étiquettes possibles.

L'automate D de l'exemple précédent est un automate complet.

N'importe quel automate non déterministe est équivalent à un automate déterministe et complet : il suffit, dans l'automate déterminisé, d'introduire un état poubelle s'il n'est pas déjà complet.

Exemple

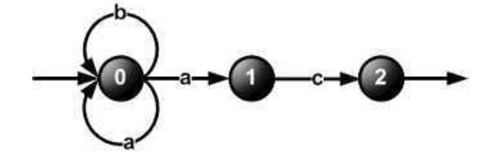


Figure 5.6

- *Déterminisation*

Tableau 5.5

	État	États résultant		
		en lisant a	en lisant b	en lisant c
(entrée)	(0)	(0,1)	(0)	–
	(0,1)	(0,1)	(0)	(2)
(sortie)	(2)	–	–	–

Nous remplaçons les traits (« pas de transition ») dans ce tableau par un nouvel état appelé poubelle (P), qui a la propriété que toutes les transitions à partir de cet état reviennent sur lui-même :

Tableau 5.6

	État	États résultant		
		en lisant a	en lisant b	en lisant c
(entrée)	(0)	(0,1)	(0)	P
	(0,1)	(0,1)	(0)	(2)
(sortie)	(2)	P	P	P
	P	P	P	P

Voici donc l'automate déterminisé complet, dans lequel (0) est resté l'état initial, et le seul état terminal reste (2) :

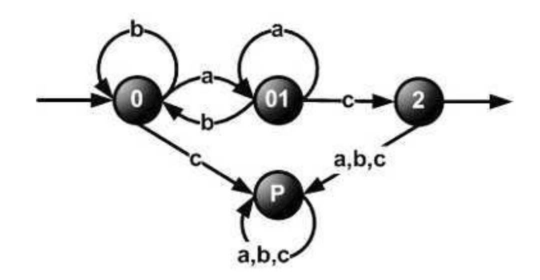
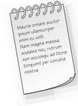


Figure 5.7

Remarque

Il n'y a pas trop de sens de parler d'un automate non déterministe complet ou pas complet. De même, même si introduire un « état poubelle » dans un automate non déterministe est possible et ne nuit pas à son fonctionnement, ceci n'est pas utile pour la déterminisation de cet automate. Normalement, on n'introduit un état poubelle si besoin est qu'une fois l'automate est devenu déterministe.



Un automate est ***accessible*** si n'importe quel état est accessible à partir d'un état initial (c.à.d. s'il existe un chemin y menant à partir d'un état initial).

Un automate est ***coaccessible*** si à partir de n'importe quel état on peut accéder à un état terminal.

Accessible + coaccessible = ***émondé***.

Un ensemble de mots X est un langage ***reconnaissable*** ssi il existe un automate fini D qui le reconnaît.

Si X est un langage reconnaissable sur l'alphabet A, on peut le reconnaître avec un automate déterministe et complet, car l'automate D figurant dans la définition du langage reconnaissable est toujours équivalent à un automate déterministe et complet F.

Écrivant $F = (Q, i, T, E)$, on a alors $w \in X$ ssi $i.w \in T$ et donc $w \notin X$ ssi $i.w \notin T$.

• *Le complément d'un langage reconnaissable*

Le complément d'un langage reconnaissable (l'ensemble de mots sur le même alphabet n'appartenant pas au langage en question) est encore reconnaissable.

Soit D un automate déterministe et complet reconnaissant le langage X.

Pour construire un automate reconnaissant le complément de X, il suffit de prendre comme ensemble d'états terminaux le complément de l'ensemble d'états terminaux de D.

Exemple

Construisons un automate sur $A = \{a, b\}$ qui reconnaît l'ensemble des mots n'ayant pas aba en facteur.

On commence par la construction d'un automate (non déterministe) reconnaissant tous les mots ayant aba en facteur.

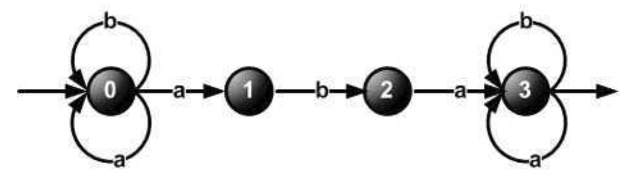


Figure 5.8

Appliquons l'algorithme de déterminisation :

Tableau 5.7

	État	États résultant	
		en lisant a	en lisant b
(initial)	(0)	(0,1)	(0)
	(0,1)	(0,1)	(0,2)
	(0,2)	(0,1,3)	(0)
(terminal)	(0,1,3)	(0,1,3)	(0,2,3)
(terminal)	(0,2,3)	(0,1,3)	(0,3)
(terminal)	(0,3)	(0,1,3)	(0,3)

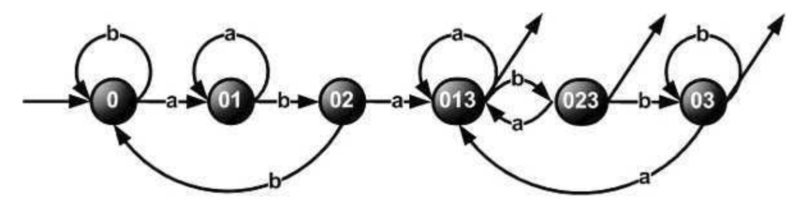


Figure 5.9

Cet automate déterministe et complet reconnaît tous les mots ayant ***aba*** en facteur. Il a trois états terminaux : (0,1,3), (0,2,3) et (0,3).

Maintenant pour obtenir un automate (lui aussi déterministe et complet) reconnaissant tous les mots **qui n'ont pas *aba*** en facteur, il suffit de faire de tous les états terminaux, des états non terminaux, et vice versa :

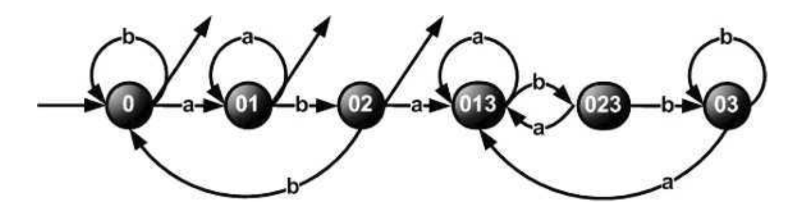


Figure 5.10



Ici, on a obtenu un automate complet sans qu'il y ait besoin de le compléter en introduisant l'état poubelle. Évidemment, ceci n'est pas toujours le cas. **Si on a affaire à un automate non complet, on n'a pas le droit de remplacer directement les états terminaux par des états non terminaux et vice versa : il faut d'abord le compléter.** Alors l'état poubelle (qui ne peut pas être terminal) devient un état terminal de l'automate reconnaissant le complément du langage.

Minimisation

Pour tout langage reconnaissable il existe ***le plus petit*** (c.-à-d. contenant le plus petit nombre d'états) automate déterministe qui le reconnaît, et il est ***unique*** : c'est l'automate minimal du langage.

• *Algorithme de minimisation par la méthode des équivalences*

Il est basé sur la notion de partitionnement de l'ensemble d'états de l'automate.

Principe Tout d'abord, si l'automate à minimiser n'est pas complet, *il faut lui ajouter l'état poubelle pour qu'il le devienne.* Sinon on risque de faire une erreur grave, qui se manifeste facilement au cas d'un automate déterministe non complet dont tous les états sont des états terminaux (essayez le voir vous-mêmes après avoir compris l'algorithme de minimisation).

On sépare tous les états de l'automate déterministe initial en deux groupes : terminaux et non-terminaux. Puis, on analyse la table des transitions en marquant vers quel groupe va chaque transition. On repartitionne les groupes selon les patterns des transitions en terme de groupes. On répète ce processus en utilisant cette nouvelle partition, et on réitère jusqu'à ce qu'on arrive à ne plus pouvoir partitionner. Les groupes restants forment l'automate minimal. On appelle souvent les itérations les étapes.

Description du processus de partitionnement itératif Soit un automate fini déterministe complet $D = (Q, i, T, E)$.

Résultat à obtenir : Un automate fini déterministe complet D' qui reconnaît le même langage que D et qui a aussi peu d'états que possible.

Méthode Construire une partition initiale Θ_0 de l'ensemble des états avec deux groupes : les états terminaux T et les états non-terminaux $Q - T$.

Procédure applicable à la partition courante Θ_i, à commencer par Θ_0 :

> POUR chaque groupe G de Θ_i FAIRE
> début
> partitionner G en sous-groupes de manière que deux états e et t de G soient dans le même sous-groupe ssi pour tout symbole $a \in A$, les états e et t ont des transitions sur a vers des états du même groupe de Θ_i ;
> /* au pire, un état formera un sous-groupe par lui-même */
> remplacer G dans par tous les sous-groupes ainsi formés ;
> on obtient la partition de l'étape suivant, Θ_{i+1} ;
> fin
> Si $\Theta_{i+1} = \Theta_i$ alors $\Theta_{final} = \Theta_i$ et continuer à l'étape 4.
> Sinon, répéter l'étape (2) avec Θ_{i+1} comme partition courante.

Choisir un état dans chaque groupe de la partition Θ_{final} en tant que représentant de ce groupe. Les représentants seront les états de l'automate fini déterministe réduit D'.

Soit e un état représentatif, et supposons que dans D il y a une transition $e \xrightarrow{a} t$.
Soit r le représentant du groupe de t (r peut être égal à t).

Alors D' a une transition de e vers r sur a ($e \xrightarrow{a} r$).

L'état initial de D' est le représentant du groupe qui contient l'état initial i de D ; les états terminaux de D' sont des représentants des états de T.

Pour tout groupe G de Θ_{final}, soit G est entièrement composé d'états de T, soit G n'en contient aucun.

Remarque

En réalité, il est parfois plus facile, au lieu de choisir un représentant, marquer les groupes finaux comme tels, ou bien les renommer par, par exemple, A, B, C ... , et remplacer dans les transitions tout état par le nom de son groupe :
Si $e \xrightarrow{a} r$, $e \in A$, $r \in B$ où les groupes $A, B \in \Theta_{final}$, alors dans l'automate minimisé on a $A \xrightarrow{a} B$ où maintenant on considère A et B comme états de l'automate minimisé. Cette remarque deviendra tout à fait claire à la fin de l'exemple suivant.

Exemple 1

Minimisons l'automate déterministe complet :

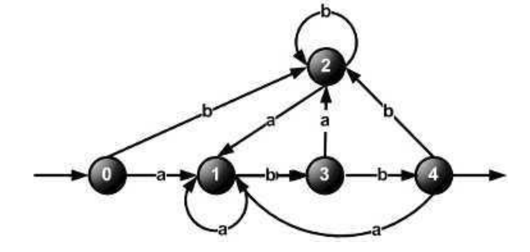


Figure 5.11

décrit par le tableau de transitions suivant :

Tableau 5.8

	État	États résultant	
		en lisant a	en lisant b
(initial)	0	1	2
	1	1	3
	2	1	2
	3	2	4
(terminal)	4	1	2

L'unique état terminal est l'état 4.
Donc, la partition initiale : $\Theta_0 = \{(0,1,2,3),(4)\}$.
Notons les groupes non terminal et terminal : $I = \{(0,1,2,3)\}$ et $II = \{(4)\}$.
On ne peut pas, évidemment, essayer de séparer le groupe II qui consiste déjà en un seul état.
On regarde donc dans quel groupe tombent les transitions à partir des états du groupe I :

Tableau 5.9

État	Groupes résultant	
	en lisant a	en lisant b
0	I	I
1	I	I
2	I	I
3	I	II

Donc, le groupe I se sépare en deux, et on a $\Theta_1 = \{(0,1,2),(3),(4)\}$.
Notons les groupes (en recyclant la notation) : $I = \{(0,1,2)\},\ II = \{(3)\},\ III = \{(4)\}$.
Maintenant essayons de séparer le groupe I en regardant où tombent les transitions à partir de ses états dans les termes de la séparation $\Theta_1 = \{(0,1,2),(3),(4)\}$.

Tableau 5.10

État	Groupes résultant	
	en lisant a	en lisant b
0	I	I
1	I	II
2	I	I

Donc, le groupe I se sépare en deux, et on a $\Theta_2 = \{(0,2),(1),(3),(4)\}$.
Notons les groupes (en recyclant la notation) :

$$I = \{(0,2)\},\ II = \{(1)\},\ III = \{(3)\},\ IV = \{(4)\}.$$

Essayons de séparer le groupe I en regardant où tombent les transitions à partir de ses états dans les termes de la séparation $\Theta_2 = \{(0,2),(1),(3),(4)\}$.

Tableau 5.11

État	Groupes résultant	
	en lisant a	en lisant b
0	II	I
2	II	I

Le groupe I ne se sépare pas, $\Theta_3$reste identique à la séparation de l'étape précédent :

$$\Theta_3 = \Theta_2 = \{(0,2), (1), (3), (4)\}.$$

Fin de la procédure itérative de séparation.
Voici les itérations qu'on a effectuées :
Étape 1 : $\Theta_{courant} = \{(0,1, 2,3), (4)\}$
$\Theta_{new} = \{(0,1, 2), (3), (4)\}$
Étape 2 : $\Theta_{courant} = \{(0,1, 2), (3), (4)\}$
$\Theta_{new} = \{(0,2), (1), (3), (4)\}$
Étape 3 : $\Theta_{courant} = \{(0,2), (1), (3), (4)\}$
$\Theta_{new} = \{(0,2), (1), (3), (4)\} = \Theta_{final}$
Passons aux représentants :

Tableau 5.12

Groupe	Représentant
I	0 ou 2
II	1
III	3
IV	4

Il devient clair maintenant qu'on peut marquer les états de l'automate minimisé soit par un représentant, soit par les chiffres I, II, III, IV suivant le groupe du dernier étape, soit par le contenu du groupe (dans ce cas, l'état initial sera marqué (0,2) ; cette dernière solution est pratique tant que le groupe consiste en peu d'états).
L'automate minimisé :

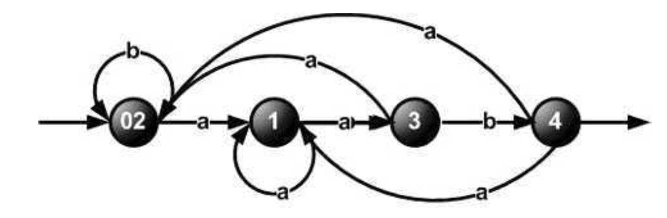


Figure 5.12

On peut maintenant poser la question suivante : pourquoi les états 0 et 2 sont resté ensemble après la minimisation ? Qu'y a-t-il de spécial concernant ces deux états par rapport aux autres ? Regardons à nouveau l'automate initial :

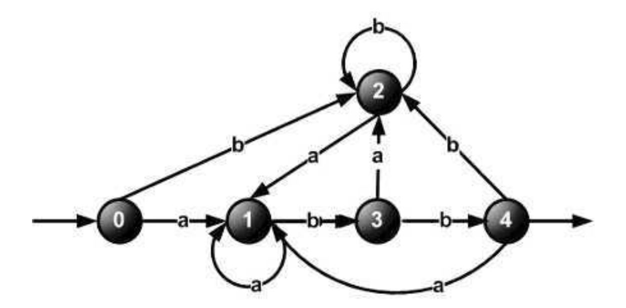


Figure 5.13

et introduisons la terminologie suivante :
on dit que la chaîne w distingue l'état s de l'état t si quand on commence dans l'état s et lit$w$ on arrive dans un état terminal, et si on commence dans t et lit w on arrive dans un état non terminal ou vice-versa.

Ici, tous les états peuvent être distingués par une chaîne ou une autre, sauf les états 0 et 2. C'est facile à voir : et à partir de 0, et à partir de 2 on arrive en 1 en lisant un nombre quelconque (y compris zéro) de b et un a; puis, comme on est dans le même état (1), il ne nous reste que les mêmes chaînes pour arriver à l'état final (ici, il y en a un seul). Donc, les états non distinguables se fondent dans un même état (en l'occurrence (0,2)) de l'automate minimisé.

En fait, on peut montrer que l'algorithme de minimisation qu'on a expliqué, est basé sur la recherche des tous les groupes qui peuvent être distingués par une chaîne ou $I = \{1\}$, $T = \{1{,}2\}$ une autre.

Exemple 2

Voici un automate déterministe complet avec deux états terminaux, avec l'alphabet A $= \{0, 1\}$:

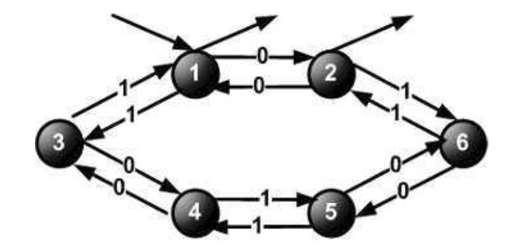


Figure 5.14

La procédure de minimisation donne :

$I = \{1\}, \quad T = \{1{,}2\}$

$\Theta_0 = \{(1,2), (3,4, 5,6)\} = (I, II\}$
$\Theta_1 = (\{(1,2).(3,6), (4,5)\} = \{I, II, III\}$ (en recyclant les chiffres romains)
$\Theta_2 = \Theta_1$
Voici l'automate minimisé :

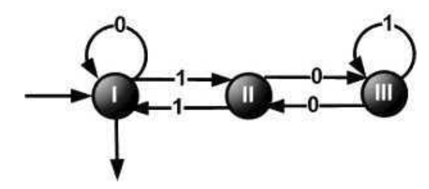


Figure 5.15

5.3.2 Automate asynchrone

Souvenons-nous que jusqu'à ce point nous avons évité **le mot vide**.

Un automate fini dans lequel certaines flèches sont étiquetées par le mot vide (« ***ε-transitions*** ») s'appelle **automate asynchrone**. L'ensemble des flèches vérifie donc $E \subset Q \times \left(A \cup \{\varepsilon\}\right) \times Q$.

Proposition : pour tout automate asynchrone, il existe un automate fini ordinaire (c.à.d. sans ε-transitions) qui reconnaît le même langage.

Comme tout automate non déterministe est équivalent à un automate déterministe, il en suit que **tout automate asynchrone est équivalent à un automate déterministe**.

Il existe un algorithme de suppressions de ε-transitions.

Pour un automate asynchrone $C = (Q, I, T, E)$ avec $E \subset Q \times \left(A \cup \{\varepsilon\}\right) \times Q$ nous allons construire un automate $B = (Q, I, T, F)$ qui ne diffère de C que par l'ensemble de ses flèches et ce, suivant une règle bien déterminée :

- par définition on a $(p.a.q) \in F$ s'il existe un chemin

$$c : p_0 \xrightarrow{\varepsilon} p_1 \xrightarrow{\varepsilon} \ldots \xrightarrow{\varepsilon} p_n \xrightarrow{a} q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow{\varepsilon} \ldots \xrightarrow{\varepsilon} q_m$$

$$\text{et } p_0 = p \wedge q_m = q$$

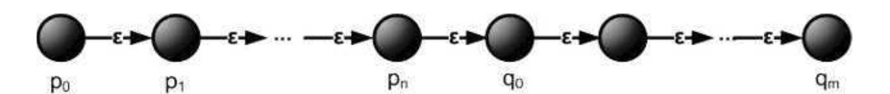


Figure 5.16

- Le nombre de ε-transitions à droite ou à gauche de la transition $p_n \xrightarrow{a} q_0$ peut être nul (dans ce cas $p_n = p_0$ou $p_n = p_0$).

Exemple

a) Prenons un automate asynchrone :

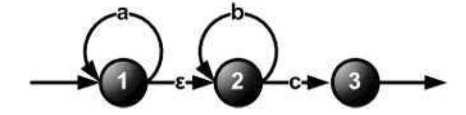


Figure 5.17 **(A1)**

Ici, suivant le raisonnement précédent, il existent les chemins : $1a1$, $1a2$, $1b2$, $1c3$, $2b2$, $2c3$.
b) Donc cet automate est équivalent à l'automate non déterministe suivant :

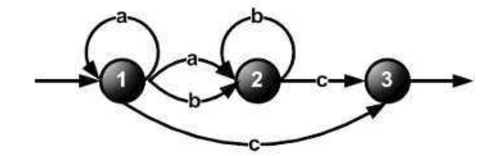


Figure 5.18 **(A2)**

c) Maintenant on peut le déterminiser :

Tableau 5.13

État	États résultant		
	en lisant a	en lisant b	en lisant c
1	1,2	2	3
1,2	1,2	2	3
2	–	2	3
3	–	–	–

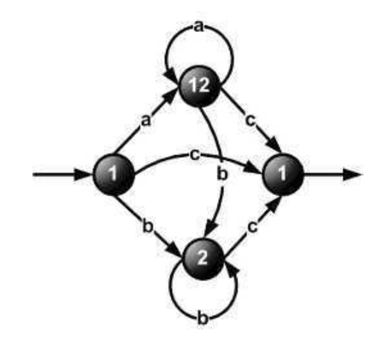


Figure 5.19 **(A3)**

ou bien, en ajoutant l'état poubelle :

Tableau 5.14

État	États résultant		
	en lisant a	en lisant b	en lisant c
1	1,2	2	3
1,2	1,2	2	3
2	P	2	3
3	P	P	P
P	P	P	P

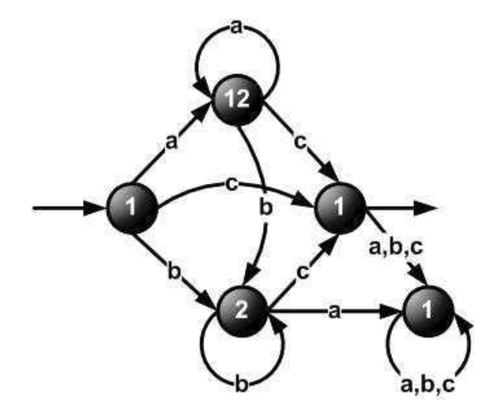


Figure 5.20 **(A4)**

d) Mais en réalité, il est plus simple de se passer de l'étape (b) et construire un automate déterministe directement à partir d'un automate asynchrone.
Redessinons l'automate (A1) :

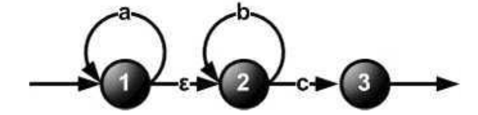


Figure 5.21 **(A1)**

L'état initial de l'automate déterministe correspondant à (A1) est (1,2), car en lisant le mot vide on arrive et en 1, et en 2. Continuons à déterminiser en marquant dans les colonnes correspondant à la lecture de chaque caractère, tous les états où on arrive après la lecture du caractère en question, c.à.d. non seulement l'état où mène la flèche étiquetée par ce caractère (disons, a), mais aussi tous les états où on peut arriver en lisant $a\varepsilon$, $a\varepsilon\varepsilon$ etc. Dans notre cas particulier, il n'y a pas de telles transitions, mais il faut systématiquement en tenir compte.

Tableau 5.15

État	États résultant		
	en lisant a	en lisant b	en lisant c
1,2	1,2	2	3
2	–	2	3
3	–	–	–

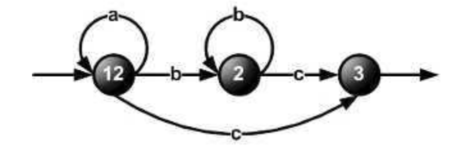


Figure 5.22 (A5)

On voit que l'automate (A5) n'est pas le même que l'automate (A3) ou (A4) ! Pourquoi ?

Parce que l'automate déterministe reconnaissant un certain langage n'est pas unique, ce n'est qu'en minimisant qu'on arrive à un automate unique. En minimisant (A4) et (A5), on doit obtenir la même chose.

Minimisons (A4) donc on rappelle la table de transitions :

Tableau 5.16

État	États résultant		
	en lisant a	en lisant b	en lisant c
1	1,2	2	3
1,2	1,2	2	3
2	P	2	3
3	P	P	P
P	P	P	P

Ici , on voit immédiatement que les états 1 et 1,2 ne se sépareront jamais !

$\Theta_0 = \{(1,(1{,}2),2,P),\ (3)\} = \{I,(3)\}$ (Utilisons une notation un peu plus intelligente pour ne pas modifier le sens des chiffres romains sans cesse)

Tableau 5.17

État	Groupes résultant		
	en lisant a	en lisant b	en lisant c
1	I	I	3
1,2	I	I	3
2	I	I	3
P	I	I	I

(5) se sépare en formant un groupe à part (c'est toujours le cas)

$\Theta_1 = \{(1,(1{,}2),2),\ (P),\ (3)\} = \{I,(P),(3)\}$

Tableau 5.18

État	Groupes résultant		
	en lisant a	en lisant b	en lisant c
1	I	I	3
1,2	I	I	3
2	P	I	3

(2) se sépare.
$\Theta_2 = \{(1,(1,2)),\ (2),\ (P),\ (3)\} = \{I,(2),(P),(3)\}$

Tableau 5.19

État	Groupes résultant		
	en lisant a	en lisant b	en lisant c
1	I	2	3
1,2	I	2	3

Le groupe (1,(1,2)) n'a aucun chance à se scinder en deux, car dès le début (1) et (1,2) ont les mêmes transitions. Donc on a terminé, et on obtient :

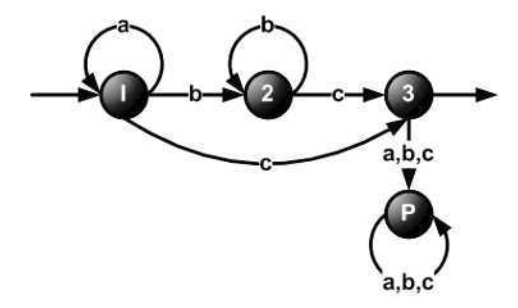


Figure 5.23

ce qui est le même automate que le résultat de compléter l'automate (A5) ! (en minimisant l'automate (A5), la seule chose qu'il reste à faire c'est de le compléter, autrement il est déjà minimal).

ÉNONCÉS DES EXERCICES

Exercice 5.1 Construire un automate fini dont les états correspondent aux situations de famille possibles d'une personne (célibataire, marié, divorcé, veuf) et dont les flèches correspondent aux changements de situation possible. Étiqueter ces flèches par M (mariage), D (divorce) et V (veuvage).

Exercice 5.2 Construire des automates finis qui reconnaissent les langages suivants :

- (a) L'ensemble des mots sur l'alphabet $\{0, 1\}$ dont l'avant dernier symbole est 0.

- (b) L'ensemble des mots sur l'alphabet {0, 1} qui commencent et qui finissent par 0. (On considère que le mot consistant en un seul 0 vérifie cette condition).
- (c) L'ensemble des mots sur l'alphabet {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, .} qui représentent en langage C des constantes numériques.
- (d) L'ensemble des mots sur l'alphabet {0, 1} qui comportent au moins une fois le motif 10 et au moins une fois le motif 01 (ces deux motifs ne sont pas obligés d'être séparés l'un de l'autre – par exemple, la séquence 010 comporte les deux motifs).
- (e) $\{a^n b^m | n + m \text{ pair}\}$

Exercice 5.3 Construire un automate fini reconnaissant les entiers écrits en base deux divisibles par cinq.

Exercice 5.4 Construire un automate fini reconnaissant les lignes contenant des commentaires d'un programme écrit en langage C. La sortie de l'automate correspond à la fin du commentaire.
On prendra comme alphabet $\{/, *, ", a\}$ où a représentera tous les caractères autres que ", * et /.
Tout commentaire commence par /* et finit par */.
Le commentaire peut contenir des / et des *, mais il ne peut pas contenir le motif */, sauf à l'intérieur d'une chaîne qui commence par " et finit par " sans contenir d'autres ". En fait, toute chaîne de caractères placée entre les guillemets doubles, est déspécialisée ; donc le nombre de guillemets doubles doit être pair et avant le commentaire, et à l'intérieur du commentaire.
La séquence /*/ n'est pas considérée comme comportant et le motif /*, et le motif */.
Exemple d'un commentaire reconnu par l'automate : /*commentaire "/*" toto"*/" **/

Exercice 5.5 Quel est le langage reconnu par l'automate suivant ? Quels états de cet automate sont inutiles ?

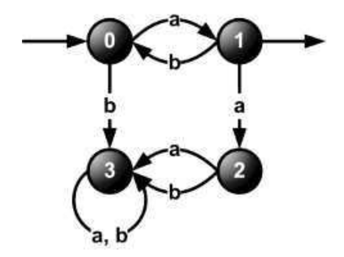


Figure 5.24

Exercice 5.6 Voici cinq automates notés A_n (ils n'ont rien à voir les uns avec les autres, ce n'est pas une séquence !). L'alphabet A = {a, b} sauf pour A_4 où il est {0,1}. Pour chacun de ces automates :

- Décrire $L(A_n)$ en langage ordinaire (sauf pour A_5).
- Caractériser A_n (dire s'il est accessible, coaccessible, émondé...)
- Calculer l'automate déterministe équivalent à A_n.

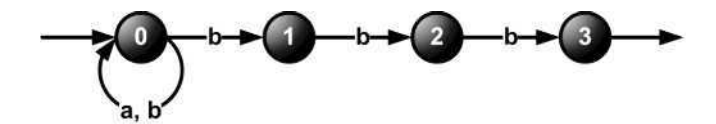


Figure 5.25 A1

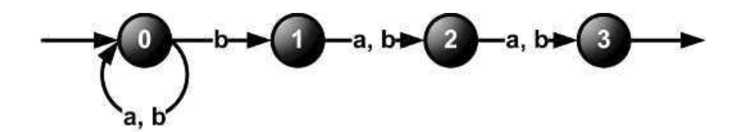


Figure 5.26 A2

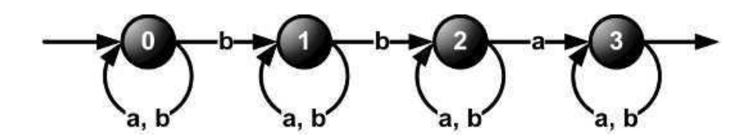


Figure 5.27 A3

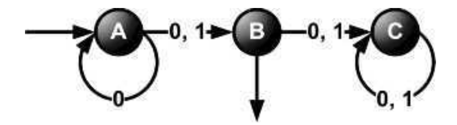


Figure 5.28 A4

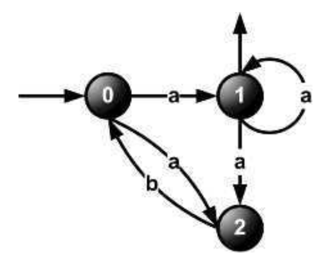
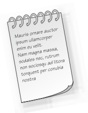


Figure 5.29 A5



(Ici, ne pas décrire $L(A_5)$ avant que l'automate ne soit déterminisé)

Exercice 5.7 Déterminiser l'automate suivant :

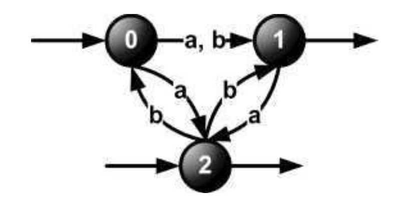


Figure 5.30

Exercice 5.8 Construire des automates déterministes qui reconnaissent les langages suivants. En déduire des automates déterministes qui reconnaissent les complémentaires de ces langages :

- (a) L'ensemble des mots sur l'alphabet $\{0, 1\}$ qui contiennent **exactement** trois fois le symbole 1.
- (b) L'ensemble des mots sur l'alphabet $\{0, 1\}$ qui contiennent **au moins** un 1.

Exercice 5.9 Construire un automate fini reconnaissant l'ensemble des mots sur l'alphabet $A = \{a,b\}$ qui **ne se terminent pas** par *abaab*. Le déterminiser et minimiser.

Exercice 5.10 Montrer que les automates déterministes calculés à l'exercice 6 sont tous minimaux sauf un. Lequel ?

Exercice 5.11 Déterminiser et minimiser les automates suivants :

- a)

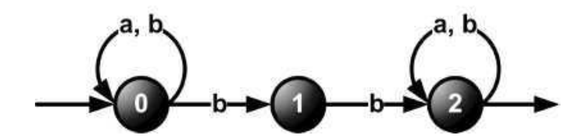


Figure 5.31

- b)

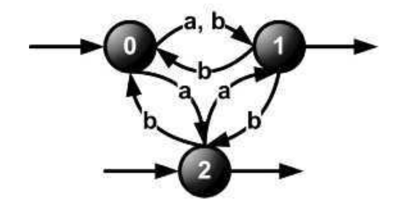


Figure 5.32

Exercice 5.12 On définit la famille d'automates suivants :

$$\widetilde{A_n} = (Q_n, I, T, E),\ n \geqslant 1$$

avec :
- $A = \{a, b\}$
- $Q_n = \{0,1, ..., n-1\}$
- $I = T = \{0\}$
- Comme flèches étiquetées par a l'ensemble de $(q.a.((q+1)\text{mod } n))$ pour $\forall q : 0 \leqslant q \leqslant n-1$
- Comme flèches étiquetées par b l'ensemble de $(q.b.0)$ et $(q.b.q)$ pour $\forall q\ :\ 1 \leqslant q \leqslant n-1$ (*attention : la première inégalité commence par 0, et la seconde, par 1*)

Dessiner $\widetilde{A_3}$ et $\widetilde{A_4}$.

Puis montrer que le déterminisé complet de $\widetilde{A_n}$ a toujours 2^n états.

Corrigés des exercices

Exercice 5.1

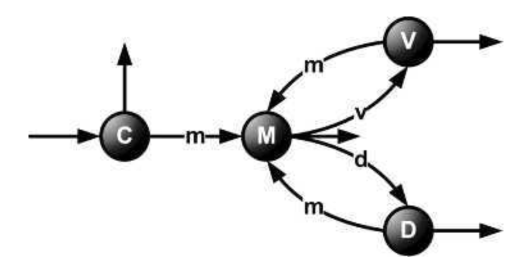


Figure 5.33 Cet automate est déterministe.

Exercice 5.2

Solution (a)

L'automate le plus simple qui reconnaît l'ensemble des mots sur l'alphabet $\{0, 1\}$ dont l'avant dernier symbole est 0, est le suivant :

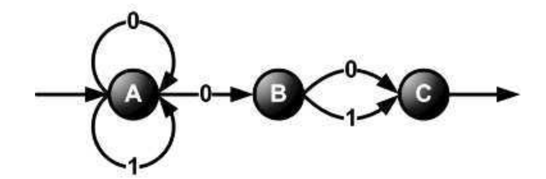


Figure 5.34 Cet automate n'est pas déterministe.

Solution (b)

L'automate le plus simple qui reconnaît l'ensemble des mots sur l'alphabet $\{0, 1\}$ qui commencent et qui finissent par 0, est le suivant :

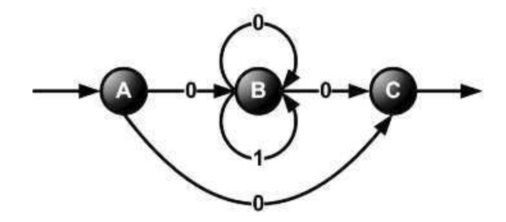


Figure 5.35 Cet automate n'est pas déterministe.

Solution (c)

Un des nombreux automates équivalents qui reconnaissent l'ensemble des mots sur l'alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, .\}$ représentant en C les constantes numériques, est le suivant :

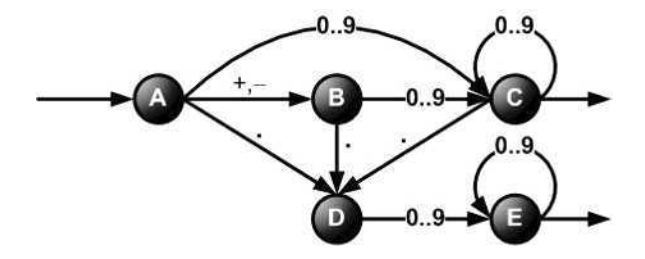


Figure 5.36 Cet automate est déterministe.

Solution (d)

Un des nombreux automates équivalents qui reconnaît l'ensemble des mots sur l'alphabet $\{0, 1\}$ comportant au moins une fois le motif 10 et au moins une fois le motif 01, est le suivant :

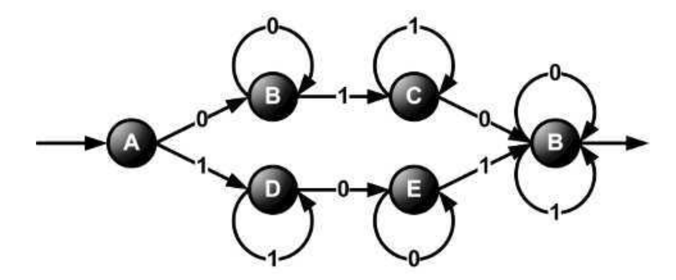


Figure 5.37 Cet automate est déterministe.

Solution (e)

Un des nombreux automates équivalents qui reconnaît le langage $\{a^n b^m | n + m \text{ pair}\}$, est le suivant :

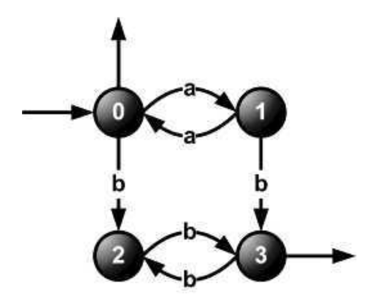


Figure 5.38 Cet automate est déterministe.

Exercice 5.3 Ajout d'un 0 à la fin d'un nombre binaire le multiplie par 2.
Ajout d'un 0 à la fin d'un nombre binaire le multiplie par 2 et lui ajoute 1.

Tableau 5.20

N	N mod 5	Ajout d'un 0 à la fin de l'écriture binaire donne N' :	N' mod 5	Ajout d'un 1 à la fin de l'écriture binaire donne N'' :	N'' mod 5
5*n*	0	10*n*	0	10*n* + 1	1
5*n* + 1	1	10*n* + 2	2	10*n* + 3	3
5*n* + 2	2	10*n* + 4	4	10*n* + 5	0
5*n* + 3	3	10*n* + 6	1	10*n* + 7	2
5*n* + 4	4	10*n* + 8	3	10*n* + 9	4

Cela se traduit par l'automate suivant :

Tableau 5.21

État	En lisant 0	En lisant 1
0	0	1
1	2	3
2	4	0
3	1	2
4	3	4

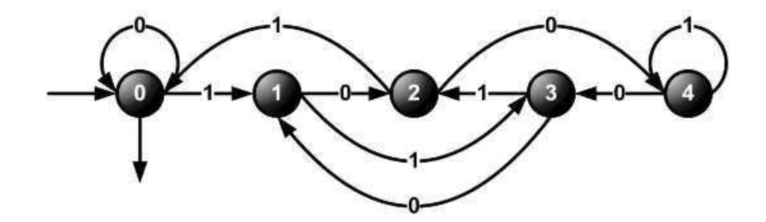


Figure 5.39

Exercice 5.4

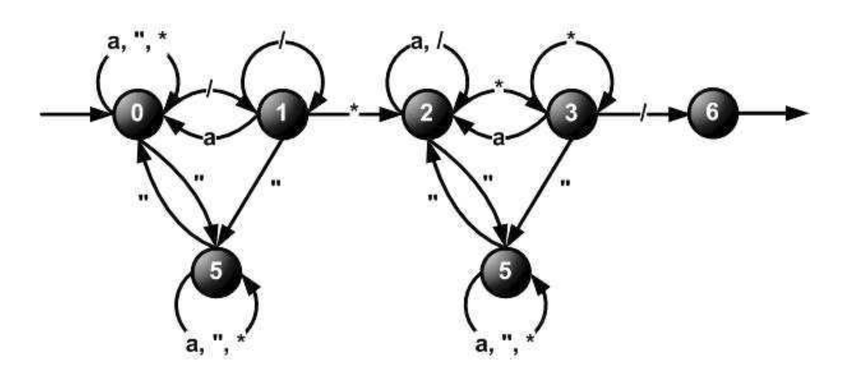


Figure 5.40

Exercice 5.5 États 2 et 3 ne mènent pas à la sortie. Ils sont donc inutiles. Le langage reconnu par cet automate est le même que le langage reconnu par :

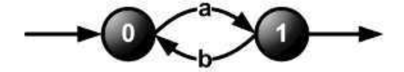


Figure 5.41

Il consiste en mots *a*, *aba*, *ababa*, *abababa*... qu'on peut écrire comme *a(ba)** ou *(ab)*a*.

Exercice 5.6

Solution A_1

- Cet automate lit tous les mots se terminant par *bbb*.
- Il est émondé.
- *Déterminisation*

Tableau 5.22

État	a	b
0	0	01
01	0	012
012	0	0123
0123	0	0123

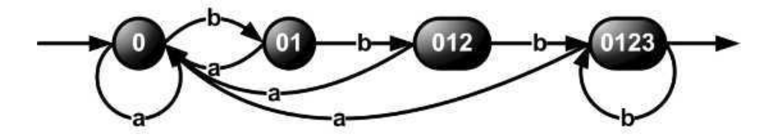


Figure 5.42 A_1

Solution A_2

- Cet automate lit tous les mots avec un *b* en 3ième position de la fin.
- Il est émondé.
- *Déterminisation*

Tableau 5.23

	État	A	b
Initial	0	0	01
	01	02	012
	012	023	0123
	02	03	013
Terminal	0123	023	0123
Terminal	023	03	013
Terminal	03	0	01
Terminal	013	02	012

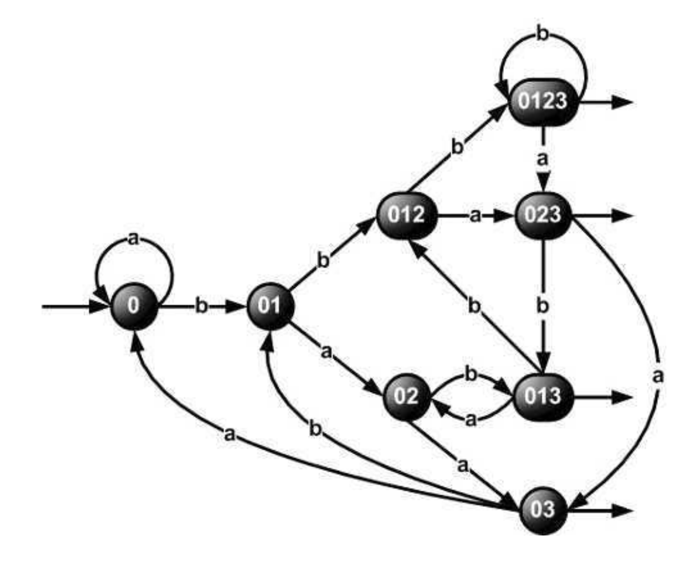


Figure 5.43

Solution A_3

- Cet automate lit tous les mots contenant au moins trois caractères ...*a*...*b*...*a*... où « ... » peut être vide.
- Il est émondé.
- • *Déterminisation*

Tableau 5.24

État	a	b
0	01	01
01	01	012
012	0123	012
0123	0123	0123

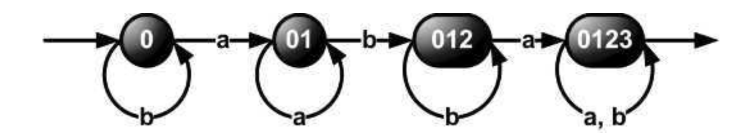


Figure 5.44

Solution A_4

- Cet automate lit tous les mots consistant en des 0 ou en des 0 suivis d'un seul 1, ou le mot 1.

- Il est accessible mais non pas coaccessible.
- *Déterminisation*

Tableau 5.25

État	0	1
A	AB	B
AB	ABC	BC
ABC	ABC	BC
B	C	C
C	C	C
BC	C	C

On voit que C est l'état poubelle.

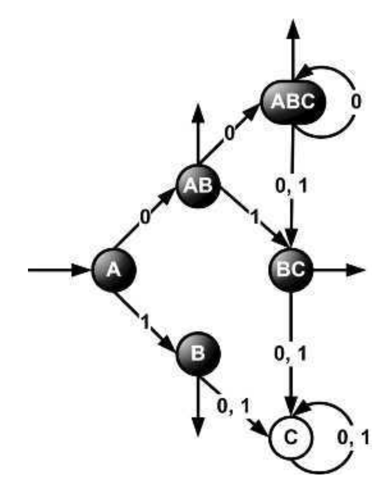


Figure 5.45

Ici on a obtenu donc un automate déterministe avec poubelle. Mais il y avait une « poubelle » dès le début – l'état C dont on ne pouvait pas revenir. On aurait pu construire d'abord un automate non déterministe plus simple qui reconnaît le même langage qu'A_4 :

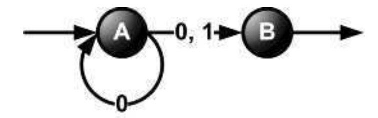


Figure 5.46

et le déterminiser. En ce faisant on arrive à un automate déterministe nettement plus simple que tout à l'heure :

Tableau 5.26

État	0	1
A	AB	B
AB	AB	B
B	P	P
P	P	P

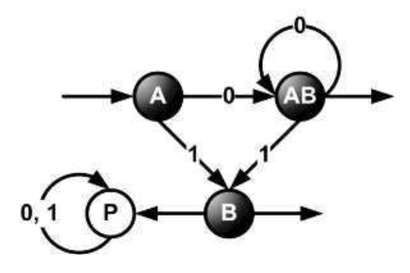
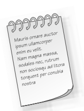


Figure 5.47



Dans l'exercice 10 on va voir que cet automate est en effet minimal.

Solution A_5

- On ne décrit pas le langage (c'est trop compliqué à ce stade).
- Il est émondé.
- *Déterminisation :*

Tableau 5.27

État	a	b
0	12	P
12	12	0

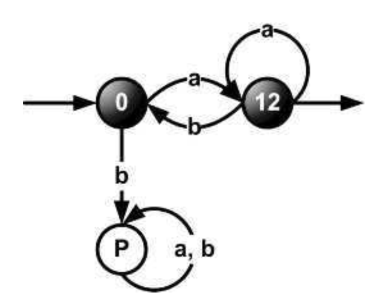


Figure 5.48

(Maintenant il devient bien plus facile de décrire$L(A_5)$ si l'on veut : il consiste en tous les mots commençant par *a*, finissant par *a*, et où tout *b* est entouré par des *a* de façon à ne jamais avoir *bb*).

Exercice 5.7

Tableau 5.28

État	a	b
02	12	01
01	12	1
12	2	01
1	2	P
2	P	01
P	P	P

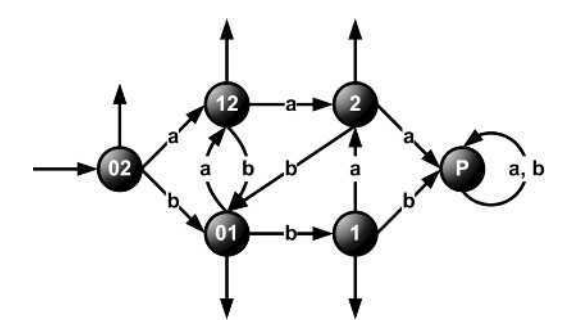


Figure 5.49

Exercice 5.8

Solution (a)

Un automate déterministe complet qui reconnaît l'ensemble des mots sur l'alphabet $\{0, 1\}$ qui contiennent exactement trois fois le symbole 1 se construit directement, sans qu'on ait besoin de construire un automate non déterministe d'abord :

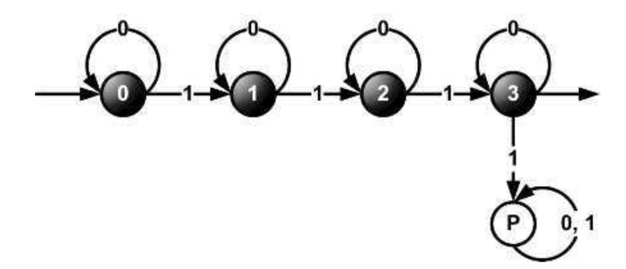


Figure 5.50

D'où l'automate reconnaissant le complément de ce langage :

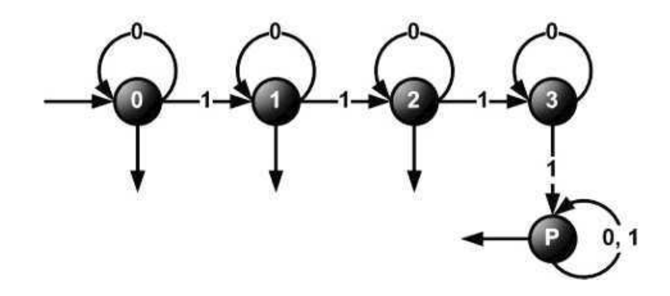


Figure 5.51



L'état P n'est plus un état poubelle, car il est un état terminal.

Solution (b)

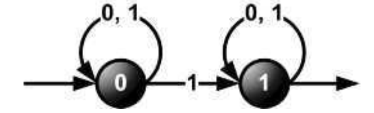


Figure 5.52

Cet automate n'est pas déterministe, il faut le déterminiser.

- *Déterminisation*

Tableau 5.29

État	0	1
0	0	01
01	01	01

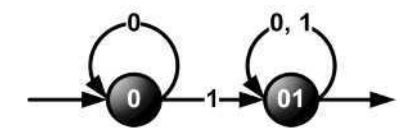


Figure 5.53

(En fait, on aurait pu dessiner cet automate dès le début).

Cet automate déterministe est complet ; donc pour construire l'automate reconnaissant le complément du langage, il suffit de modifier la position de la sortie :

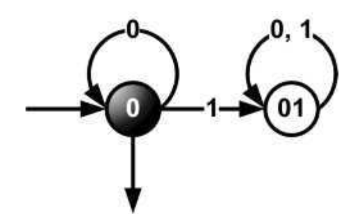


Figure 5.54 **Il est facile de voir qu'ici l'état 01 est la poubelle.**

Exercice 5.9 Nous construisons d'abord un automate qui lit tous les mots qui **se terminent** par *abaab.* Puis nous le déterminiserons et compléterons, et puis nous ferons de tous les états finaux, états non finaux et vice versa. La minimisation se fera ensuite.

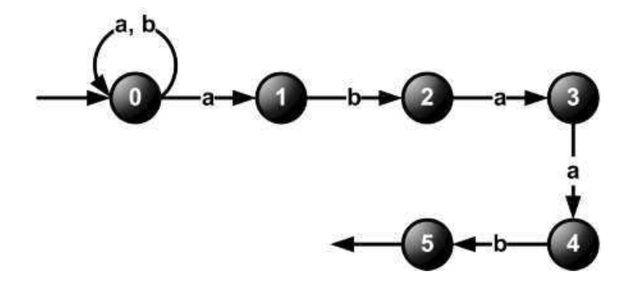


Figure 5.55

Cet automate lit tous les mots se terminant en .

- *Déterminisation*

Tableau 5.30

État	a	b
0	01	0
01	01	02
02	013	0
013	014	02
014	01	025
025	013	0

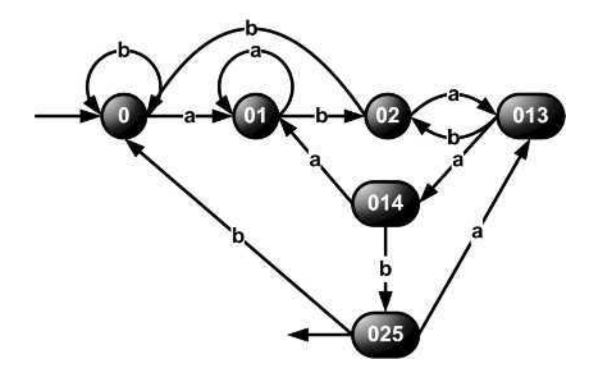


Figure 5.56

Cet automate est déjà complet, sinon il faudrait le compléter en ajoutant une poubelle.
L'automate déterministe lisant tous les mots **ne se terminant pas** par *abaab*, s'obtient comme suit :

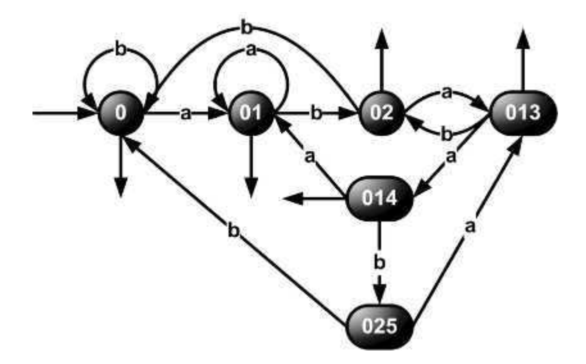


Figure 5.57

Il n'est pas minimisable, c.-à-d. qu'il est déjà minimal. Pour le voir, il faut essayer de le minimiser :
$\Theta_0 = \{I, II\}$ avec $I = \{025\}$, $II = \{0{,}01, 02{,}013, 014\}$

Tableau 5.31

État	a	b	a	b
0	01	0	II	II
01	01	02	II	II
02	013	0	II	II
013	014	02	II	II
014	01	025	II	I
025	013	0	II	II

Le groupe non terminal consiste en un seul état et n'a pas lieu à se séparer.
Le groupe terminal se sépare en deux :
$\Theta_1 = \{I, II, III\}$ avec $I = \{025\}$, $II = \{0{,}01, 02{,}013\}$, $III = \{014\}$

Tableau 5.32

État	a	b	a	b
0	01	0	II	II
01	01	02	II	II
02	013	0	II	II
013	014	02	III	II

Le groupe II se sépare en deux :
$\Theta_2 = \{I, II, III, IV\}$ avec $I = \{025\}$, II = {0, 01, 02}, III={014}, IV = {013}

Tableau 5.33

État	a	b	a	b
0	01	0	II	II
01	01	02	II	II
02	013	0	IV	II

Le groupe II se sépare en deux :
$\Theta_3 = \{I, II, III, IV, V\}$ avec $I = \{025\}$, II = {0, 01}, III = {014}, IV = {013}, V={02}

Tableau 5.34

État	a	b	a	b
0	01	0	II	II
01	01	02	II	V

$\Theta_4 = \Theta_3$; tout les états se sont séparés ; l'automate était minimal.

Exercice 5.10

Solution A_1

Reprenons l'automate déterministe complet A_1 :

Tableau 5.35

État	a	b
0	0	01
01	0	012
012	0	0123
0123	0	0123

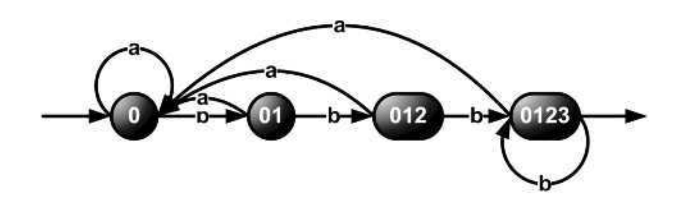


Figure 5.58

• *Minimisation*

$Q_0 = \{I,\ II\}$ avec $I = \{0,\ 01,\ 012\}$, $II = \{0123\}$.

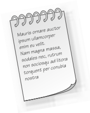
Tableau 5.36

État	a	B	A	b
0	0	01	I	I
01	0	012	I	I
012	0	0123	I	II

Le groupe I se sépare en deux. Modifiant la notation, on a :
$Q_1 = \{I,\ II,\ III\}$ avec $I = \{0,\ 01\}$, $II = \{012\}$, $III = \{0123\}$.

Tableau 5.37

État	a	B	a	b
0	0	01	I	I
01	0	012	I	II



Les deux états du groupe I se sont séparés, donc tous les états se sont séparés – l'automate était minimal dès le début.

Solution A_2

Reprenons l'automate déterministe complet A_2 mais modifions la notation des états pour plus de clarté :

Tableau 5.38

				En renommant les états		
	État	**a**	**b**	**État**	**a**	**b**
Initial	0	0	01	A	A	B
	01	02	012	B	D	C
	012	023	0123	C	F	E
	02	03	013	D	G	H
Terminal	0123	023	0123	E	F	E
Terminal	023	03	013	F	G	H
Terminal	03	0	01	G	A	B
Terminal	013	02	012	H	D	C

(Le dessin n'est pas trop utile pour cet automate.)

- *Minimisation*

Θ_0 = {I, II} avec I = {A, B, C, D}, II = {E, F, G, H}.

Tableau 5.39

	État	**a**	**b**	**a**	**b**
Groupe non terminal	A	A	B	I	I
	B	D	C	I	I
	C	F	E	II	II
	D	G	H	II	II
Groupe terminal	E	F	E	II	II
	F	G	H	II	II
	G	A	B	I	I
	H	D	C	I	I

Θ_1 = {I, II, III, IV} avec I = {A, B}, II={C, D}, III={E, F}, IV={G, H}.

Tableau 5.40

État	a	b	A	b
A	A	B	I	I
B	D	C	II	II
C	F	E	III	III
D	G	H	IV	IV
E	F	E	III	III
F	G	H	IV	IV
G	A	B	I	I
H	D	C	II	II

Tous les états se sont séparés, l'automate était minimal dès le début.

Solution A_3

Reprenons l'automate déterministe complet A_3 :

Tableau 5.41

État	a	b
0	01	01
01	01	012
012	0123	012
0123	0123	0123

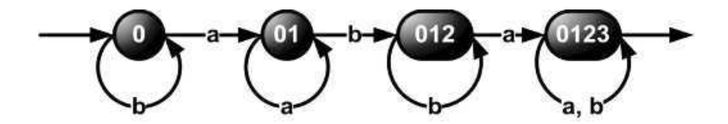


Figure 5.59

- *Minimisation*

$\Theta_0 = \{I, II\}$ avec $I = \{0, 01, 012\}$, $II = \{0123\}$

Tableau 5.42

État	a	b	a	b
0	01	01	I	I
01	01	012	I	I
012	0123	012	II	I

L'état 012 est sorti du groupe I. Nouveaux groupes :
$\Theta_1 = \{I, II, III\}$ avec $I = \{0, 01\}$, $II = \{012\}$, $III = \{0123\}$.

Tableau 5.43

État	a	b	a	B
0	01	01	I	I
01	01	012	I	II

Tous les états se sont séparés, l'automate était minimal dès le début.

Solution A_4

Reprenons l'automate déterministe complet A_4 :

Tableau 5.44

État	0	1
A	AB	B
AB	ABC	BC
ABC	ABC	BC
B	C	C
C	C	C
BC	C	C

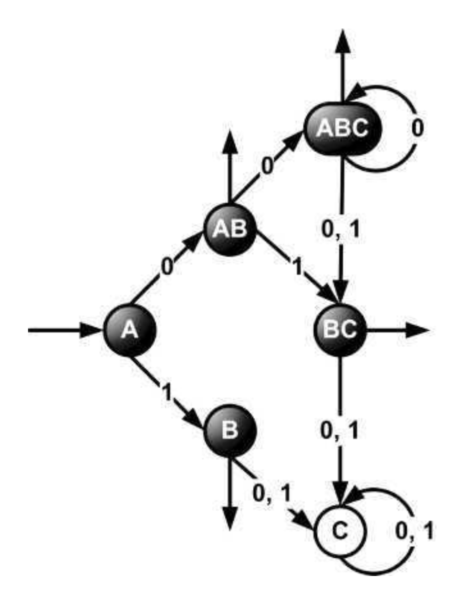


Figure 5.60

- *Minimisation*

$\Theta_0 = \{I, II\}$ avec $I = \{A, C\}$, $II = \{AB, ABC, BC, B\}$.

Tableau 5.45

	État	0	1	0	1
Groupe non terminal	A	AB	B	II	II
	C	C	C	I	I
Groupe terminal	AB	ABC	BC	II	II
	ABC	ABC	BC	II	II
	BC	C	C	I	I
	B	C	C	I	I

On peut voir immédiatement que les états A et C se séparent, et que le groupe II se divise en deux groupes, qui ne pourront plus se séparer. On obtient donc les états de l'automate minimisé : A, (B, BC), (AB, ABC), et C.

Graphiquement, cet automate minimal se présente comme :

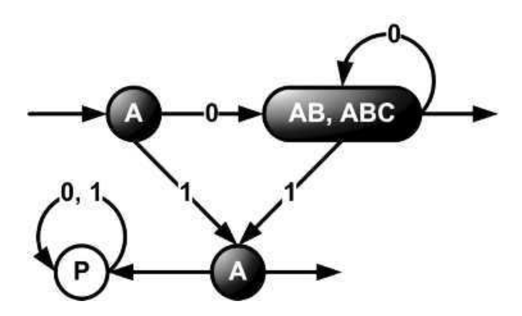


Figure 5.61

où l'état C sert comme l'état poubelle. L'automate déterministe correspondant à A_4 n'était donc pas minimal. On voit qu'en le minimisant on est arrivé au même automate qu'on a obtenu en déterminisant l'automate A_4 initial (non déterministe) duquel on a coupé sa « poubelle ».

Remarque

On rappelle encore une fois que la « poubelle » d'un automate non déterministe (l'état ou le groupe d'états dont on ne peut pas allez vers la ou les sortie(s) n'est pas la même chose que la poubelle de l'automate déterministe qui en résulte après la déterminisation.

La seule utilité de la « poubelle » d'un automate non déterministe consiste en la possibilité de la couper pour simplifier la déterminisation.

Solution A_5

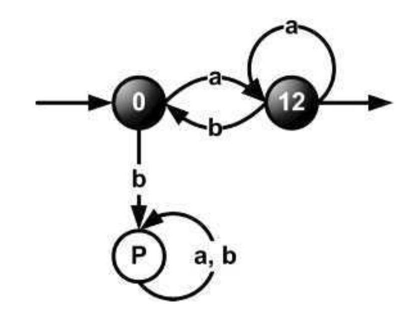


Figure 5.62 A_5

- *Minimisation*

$\Theta_0 = \{I, II\}$ avec $I = \{0, P\}$, $II = \{12\}$.

Tableau 5.46

État	a	b	a	b
0	12	P	II	I
P	P	P	I	I
12	12	0	II	I

Les deux états du groupe I se sont séparés ; donc tous les états se sont séparés, l'automate était minimal dès le début.

Exercice 5.11

Solution (a)

- *Déterminisation*

Tableau 5.47

État	a	b
0	0	01
01	0	012
012	02	012
02	02	012

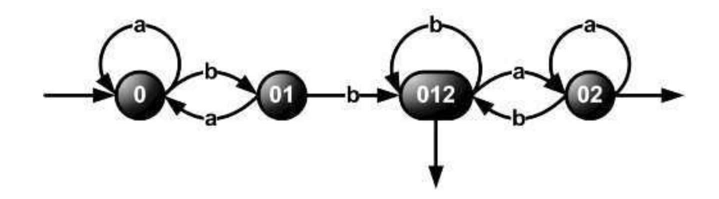


Figure 5.63

- *Minimisation*

$\Theta_0 = \{I, II\}$ avec $I = \{0, 01\}$, $II = \{012, 02\}$.

Tableau 5.48

État	a	b	a	b
0	0	01	I	I
01	0	012	I	II
012	02	012	II	II
02	02	012	II	II

Le groupe II ne peut pas se séparer. Le groupe I se sépare en 0 et 01.
Le résultat :

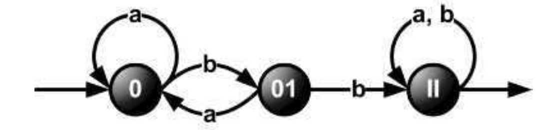


Figure 5.64

Solution (b)

- *Déterminisation*

Tableau 5.49

			En modifiant les étiquettes des états		
État	**a**	**b**	**État**	**a**	**b**
02	12	01	A	B	D
01	12	012	D	B	E
12	1	02	B	C	A
012	12	012	E	B	E
1	P	02	C	P	A
P	P	P	P	P	P

Tous les états sont des états finaux sauf l'état poubelle.

Tableau 5.50

État	a	b	a	b
A	B	D	II	II
D	B	E	II	II
B	C	A	II	II
E	B	E	II	II
C	P	A	I	II

- *Minimisation*

$\Theta_0 = \{I, II\}$ avec I = {P}, II = {A, B, C, D, E}.
Nouveaux groupes d'états : I = {P}, II = {C}, III = {A, B, D, E}.

Tableau 5.51

État	a	b	a	b
A	B	D	III	III
D	B	E	III	III
B	C	A	II	III
E	B	E	III	III

Nouveaux groupes d'états : I = {P}, II = {C}, III = {B}, IV = {A, D, E}.

Tableau 5.52

État	a	b	a	b
A	B	D	III	IV
D	B	E	III	IV
E	B	E	III	IV

Le groupe IV ne se sépare pas.
Le résultat :

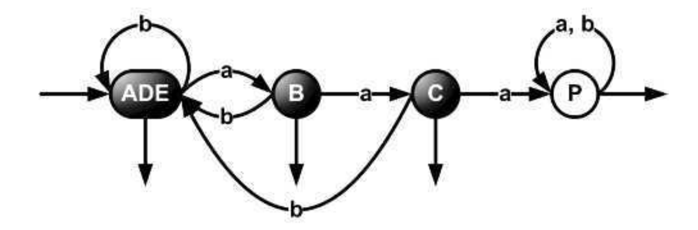


Figure 5.65

Exercice 5.12

- $\tilde{A}_3$

états 0, 1, 2;
flèches 0.a.(1 mod 3) = 0.a.1 ; 1.a.(2 mod 3) = 1.a.2 ; 2.a.(3 mod 3) = 2.a.0 ;
1.b.0; 1.b.1; 2.b.0; 2.b.2.

Tableau 5.53

État	a	b
0	1	–
1	2	0,1
2	0	0,2

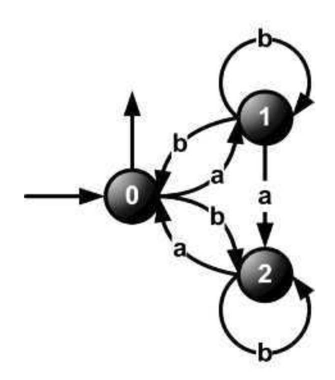


Figure 5.66

- $\tilde{A}_4$

états 0, 1, 2, 3;
flèches 0.a.(1 mod 4) = 0.a.1 ; 1.a.(2 mod 4) = 1.a.2 ;
2.a.(3 mod 4) = 2.a.3 ; 2.a.(4 mod 4) = 2.a.0 ;
1.b.0; 1.b.1; 2.b.0; 2.b.2; 3.b.0; 3.b.3.

Tableau 5.54

État	a	B
0	1	–
1	2	0,1
2	3	0,2
3	0	0,3

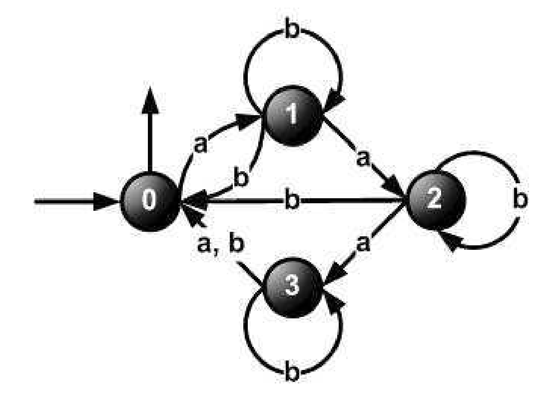


Figure 5.67

- $\tilde{A}_n$

Tableau 5.55

État	a	B
0	1	–
1	2	0,1
2	3	0,2
3	4	0,3
...	...	...
$n-1$	0	0, $n-1$

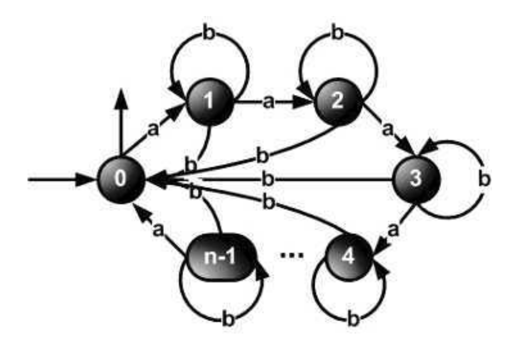


Figure 5.68

- *Déterminisation*

Tableau 5.56

État	**a**	**B**
0	1	–
1	2	01
2	3	02
3	4	03
...	...	...
$n-1$	0	$(0, n-1)$
01	12	01
02	13	02
...	...	...
$(0, n-1)$	01	$(0, n-1)$
12	23	012
13	24	013
...	...	...
$(1, n-1)$	02	$(0,1, n-1)$
...	...	...

Ici figurent toutes les combinaisons possibles de m états parmi n c'est-à-dire : $1 \leqslant m \leqslant n$.

Il y a $\sum_{m=0}^{n} C_n^m = 2^n - 1$ de tels états. Si l'on y ajoute l'état poubelle (où mène la flèche étiquetée b partant de l'état 0), pour que l'automate soit complet, on obtient 2^n états.

BIBLIOGRAPHIE

BAYNAT B., CHRÉTIENNE P., HANEN C., KEDAD-SIDHOUM S., MUNIER-KORDON A. et PICOULEAU C. – *Exercices et problèmes d'algorithmique* 2e édition. Dunod, Paris, 2007.

CARREZ C. – *Des structures aux bases de données*. Dunod, Paris, 1994.

CORMEN T., LEISERSON C., RIVEST R. et STEIN C. – *Introduction à l'algorithmique* 3e édition. Dunod, Paris, 2010.

FROIDEVAUX C., GAUDEL M.-C. et SORIA M. – *Types de données et algorithmes*. Ediscience International, Paris, 1994.

GUYOT J. et VIAL C. – *Arbres, tables et algorithmes*. Eyrolles, Paris, 1992.

LUC BOUGE L., KENYON C., MULLER J.-M. et ROBERT Y. – *Algorithmique. Exercices corrigés*. Ellipses, Paris, 1993.

INDEX

L

M

N

O

P

R

S

T

V